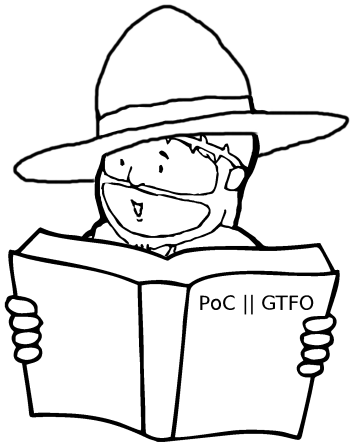
TRACT
de la
SOCIÉTÉ SECRÈTE
de
POC || GTFO
sur
L'ÉVANGILE DES MACHINES ÉTRANGES
et autres
SUJETS TECHNIQUES
par le prédicateur

PASTEUR MANUL LAPHROAIG

pastor@phrack.org



27 June 2014

MONTRÉAL:
Published by the Tract Association of POC||GTFO and Friends,
And to be Had from Their Street Prophet,
Laphroaig, at the Corner of
Rue Ste-Catherine and Rue Jeanne-Mance
Or on the Intertubes as `pocorgtfo04.pdf`.

No 0x04 Самиздат

Legal Note: Permission to use all or part of this work for personal, classroom or any other use is *NOT* granted unless you make a copy and pass it to a neighbor without fee. Just as Saint Leibowitz of Utah and his merry band of bookleggers defended their hoard from the bonfires of the Simplification, you might one day need to defend your seeds of 0day from Chris Soghoian and the ACLU's—and who could imagine ACLU in that corner?—Anti-0day-Initiative. Best of luck!

Reprints: This issue is published through samizdat as `pocorgtfo04.pdf`. While the recently successful Auernheimer appeal didn't explicitly legalize enumerating integers, you might now feel safe in counting upward from `pocorgtfo00.pdf` to get our other issues. Those who aren't as brave can run `unzip pocorgtfo04.pdf` without fear of legal repercussions.

Technical Note: Like many of our prior issues, this one is a polyglot. As a PDF, it renders to the document that you are now reading. As a ZIP, it contains our prior issues and some of that good, old-timey mythology. As a Truecrypt volume, its contents is a mystery, but "123456" might not have been the best choice of a password.

Not a .txt: We've been repeatedly asked to release as a 7-bit clean ASCII textfile, and while we too love textfiles, we find this to be terribly unneighborly. Do you motherless children show up at a concert to scream, "Shut up and play the single!"? Verily, I tell you, don't be unneighborly! When you show up at a concert, scream "Play the song that you practiced!" and enjoy the show!

Boss	Reverend Doctor Pastor Manul Laphroaig
Dept. of PHY	Michael Ossmann
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Funky File Formats Polyglot	Ange Albertini
Minister of Spargelzeit Weights and Measures	FX

1 Call to Worship

Neighbors, please join me in reading this fifth issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first four issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, or the fourth in Heidelberg. This fifth issue is written for the fine neighbors at Recon in Montréal.

We begin in Section 2, where Pastor Laphroaig presents his first epistle concerning the bountiful seeds of 0day, from which all clever and nifty things come. The preacherman tells us that the *mechanism*—not the target!—is what distinguishes the interesting exploits from the mundane.

In Section 3, Shikhin Sethi presents the first in a series of articles on the practical workings of X86 operating systems. You'll remember him from his prior boot sectors, such as Tetranglix in PoC‖GTFO 3:8 and Wódscipe, a 512-byte Integrated Development Environment for Brainfuck and ///. This installment describes the A20 address line, virtual memory, and recursive page mapping.

The first of two 6502 articles in this issue, Section 4 describes Peter Ferrie's patch to rebuild Prince of Persia to remove copy protection and fit on a single, two-sided 16-sector floppy disk. (Artwork in this section advertises the brilliant novella Prince of Gosplan by Виктор Пелевин. You should read it.)

The author of Section 5 provides a quick introduction to fuzzing with his rewrite of Sergey Bratus and Travis Goodspeed's Facedancer framework for USB device emulation.

In Section 6, Natalie Silvanovich continues the Tamagotchi hacking that you read about in PoC‖GTFO 2:4. This time, there's no software vulnerability to exploit; instead, she loads shellcode into the chip's memory and glitches the living hell out of its power supply with an AVR. Most of the time, this causes a crash, but when the dice are rolled right, the program counter lands on the NOP sled and the shellcode is executed!

In Section 7, Evan Sultanik presents a provably plausibly deniable cryptosystem, one in which the ciphertext can decrypt to multiple plaintexts, but also that the file's creator can deny ever having *intended* for a particular plaintext to be present.

In Section 8, Deviant Ollam shares a forgotten trick for modifying normal locks with a tap and die to make them pick resistant.

In Section 9, Travis Goodspeed presents an introductory tutorial on chip decapsulation and photography. Please research and follow safety procedures, as chemical accidents hurt a lot more than a core dump.

In Section 10, Colin O'Flynn exploits a pin-protected external hard disk and a popular AVR bootloader using timing and simple power analysis.

In Sections 11 and 12, our own Funky File Formats Polygot Ange Albertini shows how to hide a TrueCrypt volume in a perfectly valid PDF file so that PDF readers don't see it, and how to attach feelies ZIPs to PDF files so that Adobe tools do see them as legitimate PDF attachments. (Yes, Virginia, there is such a thing as a PDF attachment!)[1]

In Section 13, our Poet Laureate Ben Nagy presents his Ode to ECB accompanied by one of Natalie Silvanovich's brilliant public service announcements. Don't let your penguin show!

Finally, in Section 14, we do what churches do best and pass around the donation plate. Please contribute any nifty proofs of concept so that the rest of us can be enlightened!

One last thing before you dig in. This issue is brought to you by Merchants of PoC. Are you a Merchant of PoC, neighbor? Have you what it takes to follow the Great PoC Road, bringing the exotic treasures of Far and Misunderstood Parts to your neighborhoods? Or are you a Merchant of Turing-complete Death and Cyber-bullets? Fret not, neighbor: the only Merchants we fear are the Merchants of Ignorance, who seek to ban or control what they don't understand, and know not the harm they cause to the trade of Knowledge and Understanding.

[1] *So now you can put your attachments inside your attachments—but I digress.* –PML

2 First Epistle Concerning the Bountiful Seeds of 0Day

by Manul Laphroaig, Merchant of Dead Trees

Dearly Beloved,

Are the last days of 0day upon us? Is 0day becoming so sparse as to grace the very few, no matter how many of the faithful strive for its glory? Not so.

For what is the seed of 0day? Is it not a nugget of understanding what those of little faith ignore as humdrum? Is it not liberating the computing power of mechanisms unnoticed by those who use them daily? Is it not programming of machines presumed to be set in stone or silicon?

Verily, when the developer herds understand the tools that drive them to their cubicled pastures every day, then shall the 0day be depleted—but not before. Verily, when every tender of academic pigeonholes reads the papers he reviews and demands to see their source, then might the 0day begin to deplete—but not before.

For how can the sum of programs grow faster than St. Moore foresaw without increasing the sum of 0day? Have we prophets and holy ones who can cure the evil of using tools without understanding? Have layers of abstractions stopped breeding blind reliance? Verily, on such sand new castles are being erected even now.

So, beloved brethren, seek after 0day wherever and whenever the idolaters say "this just works" or "you don't need to understand this to write great code" or yet "write once, run anywhere." Most of all, look for it where the holy PEEK and POKE are withheld from those who crave them—for no righteousness can survive there, and the blind there are leading the blind to the pits of eternal pwnage.

Similarly, pay no attention to the target of an exploit. The *mechanism*, not target, is where an exploit's cleverness lies. Verily, the target, the pwnage, and the press release are all just a side show. When the neighbors ask you about BYOD, rebuke them like this: "It is not my job to sell you a damned iPad!"

So preach this good news to all your neighbors, and to their neighbors:

> If the 0day in your familiar pastures dwindles, despair not! Rather, bestir yourself to where programmers are led astray from the sacred Assembly, neither understanding what their programming languages compile to, not asking to see how their data is stored or transmitted in the true bits of the wire. For those who follow their computation through the layers shall gain 0day and pwn, and those who say "we trust in our APIs, in our proofs, and in our memory models and need not burden ourselves with confusing engineering detail that has no scientific value anyhow" shall surely provide an abundance of 0day and pwnage sufficient for all of us.

Go now in peace and pwnage,
—PML

3 This OS is a Boot Sector

by Shikhin Sethi, Merchant of 3.5" Niftiness

Writing an Operating System is easy. Explaining how to write one isn't. Most introductory articles on the same obfuscate the workings of the necessary components of an OS with design paradigms the writers feel best complement the OS. This article, the first in my PoC||GTFO series on just how a modern OS works, is different—it tries to properly, yet succinctly, explain all the requisite components of an OS—in 512 bytes per article.

The magic begins with the processor starting execution on reset at the linear address `0xFFFFFFF0`. This location contains a jump to the Basic Input/Output System (BIOS) code, which starts with the Power On Self Test (POST), followed by initialization of all requisite devices. In a predetermined order, the BIOS then checks for any bootable storage medium in the system. Except for optical drives, a bootable disk is indicated via a 16-bit 0xAA55 identifier at the 510-byte mark (end of first 512-byte sector).[2]

If a bootable medium is found, the first sector is loaded at the linear address `0x7C00` and jumped to. If none is found, the BIOS lovingly displays "Operating System not found."[3]

3.1 Real Mode

The first ancestor of today's x86 architecture was the 8086, introduced in 1978. The processor featured no memory protection or privilege levels. By 1982, Intel had designed and released the 80286, which featured hardware-level memory protection mechanisms, among other features. However, to maintain backward compatibility, the processor started in a mode compatible with the 8086 and 80186, known as *real mode*. (Feature wise, the mode lacks realness on all accounts.)

Real mode features a 20-bit address space and limited segmentation. The mode featuring memory protection and a larger address space was called the *protected mode*.

Note that the 16-bit protected mode introduced with the 80286 was enhanced with the 80386 to form 32-bit protected mode. We will be targeting only the latter.

3.2 Segmentation

The 8086 had 16-bit registers, which were used to address memory. However, its address bus was 20-bit. To take advantage of its full width and address the entire 1MiB physical address space, the scheme of 'segmentation' was devised.

In real-mode segmentation, 16-bit segment registers are used to derive the linear address. The registers CS, DS, SS, and ES point to the current code segment, data segment, stack segment respectively, with ES being an 'extra' segment.

The 80386 introduced the FS and GS registers as two more additional segment registers.

[2] `0xAA55` is representable as `0b1010101001010101`. The alternating bit pattern, with `0x55` being an inversion of `0xAA`, was taken as an insurance against even extreme controller failure. The same identifier is also used in other parts of the BIOS interface.

[3] There is no deep reason behind `0x7C00` being the load address. This is how programming usually works (and standards proliferate).

The 16-bit segment selector in the segment register yields the 16 significant bits of the 20-bit linear address. A 16-bit offset is added to this segment selector to yield the linear address. Thus, an address of the form:

$$(Segment) : (Offset)$$

can be interpreted as,

$$(Segment << 8) + Offset$$

This, however, can yield multiple (Segment):(Offset) pairs for a linear address. This problem persists during boot time, when the BIOS hands over control to the linear address `0x7C00`, which can be represented as either 0x0000:0x7C00 or `0x07C0:0x0000`. (Even the very first address the processor starts executing at reset is similarly ambiguous. In fact, 8086 and 80286 placed different values into CS and IP at reset, `0xFFFF:0x0000` and `0xF000:0xFFF0` respectively.) Therefore, our bootloader starts with a far jump to reset CS explicitly, after which it initializes other segment registers and the stack.

```
; 16-bit, 0x7C00 based code.
org 0x7C00
bits 16

; Far jump, reset CS to 0x0000.
; CS cannot be set via a 'mov', and requires a far jump.
start:
    jmp 0x0000:seg_setup

seg_setup:
    xor ax, ax
    mov ds, ax
    mov ss, ax
```

Stack

The x86 also offers a hardware stack (full-descending). SS:(E)SP points to the top of the stack, and the instructions push/pop directly deal with it.

```
    ; Start the stack from beneath start (0x7C00).
    mov esp, start
```

Flags

A direction flag in the (E)FLAGS register controls whether string operations decrement or increment their source/destination registers. We clear this flag explicitly, which implies that all source/destination registers should be incremented after string operations.

```
    ; Clear direction flag.
    cld
```

The A20 Line

On the original 8086, the last segment started at `0xFFFF0` (segment selector = 0xFFFF). Thus, with offset greater than `0x000F`, one could potentially access memory beyond the 1MiB mark. However, having only 20 addressing lines, such addresses wrapped around to the 0MiB mark. An access of `0xFFFF:0x0010` would yield an access to `0x0000` (wrapped around from `0x10000`) on the 8086.

The 80286, however, featured twenty-four address bits. Delighted hackers, on the other hand, had already exploited the wrap-around of addresses on the 80(1)86 to its fullest extent. Intel maintained backwards compatibility by introducing a software programmable gate to enable or disable the twenty-first addressing line (called the A20 line), known as the A20 gate. The A20 gate was disabled on-boot by the BIOS.

```
; Read the 0x92 port.
in al, 0x92
; Enable fast A20.
or al, 2
; Bit 0 is used to specify fast reset, 'and' it out.
and al, 0xFE
out 0x92, al
```

3.3 Protected mode

Segmentation Revisited

The introduction of protected mode featured an extension to the segmentation model, to allow rudimentary memory protection. With that extension, each segment register contains an offset into a table, known as the global descriptor table (GDT). The entries in the table describe the segment base, limit, and other attributes—including whether code in the segment can be executed, and what privilege level(s) can access the segment.

At the same time, Intel introduced paging. The latter was much easier to use for fine-grained control and different processes, and quickly superseded segmentation. All major operating systems setup 'linear' segmentation where each segment is a one-on-one mapping of the physical address space, after which they ignore segmentation.

As paging was extended to cover most cases, segmentation was left with only an empty shell of its former glory. However, it inspired OpenWall's non-executable stack patch and PaX's SEGMEXEC—both of which couldn't have been implemented with vanilla x86 paging.

Note that the new segment selectors are only valid for 32-bit protected mode, and we'll reload them after the switch to that mode.

```
; Disable interrupts.
cli
; Load the GDTR - the pointer to the GDT.
lgdt [gdtr]

; The GDT.
gdt:
    ; The first entry in the GDT is supposed to be a
    ; null entry, but we'll substitute it with the
    ; 'pointer to gdt'.
    gdtr:
        ; Size of GDT - 1.
        ; 3 entries, each 8 bytes.
        dw (0x8 * 3) - 1
        ; Pointer to GDT.
        dd gdt
        ; Make it 8 bytes.
        dw 0x0000

    ; The code entry.
    dw 0xFFFF           ; First 16-bits of limit.
```

```
    dw 0x0000           ; First 16-bits of base.
    db 0x00             ; Next 8-bits of base.
    db 0x9A             ; Read/writable, executable, present.
    db 0xCF             ; 0b11001111.
                        ; The least significant four bits are next four bits of
                        ; limit.
                        ; The most significant two bits specify that this is for
                        ; 32-bit protected mode, and that the 20-bit limit is in
                        ; 4KiB blocks. Thus, the 20-bit 0b11111111111111111111
                        ; specifies a limit of 0xFFFFFFFF.
    db 0x00             ; Last 8-bits of base.

    ; The data entry.
    dw 0xFFFF, 0x0000
    db 0x00
    db 0x92             ; Read/writable, present.
    db 0xCF
    db 0x00
```

No More Real (Mode)

The switch to protected mode is relatively easy, involving merely setting a bit in the CR0 register and then reloading the CS register to specify 32-bit code.

```
    mov eax, cr0
    or eax, 1           ; Set the protection enable bit.
    mov cr0, eax
    jmp 0x08:protected_mode

bits 32
protected_mode:
    ; Selector 0x10 is the data selector offset.
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov ss, ax
```

3.4 Paging

"Paging is called paging because you need to draw it on pages in your notebook to succeed at it."
—Jonas 'Sortie' Termansen

Virtual Memory

The concept of virtual memory is to have per-process virtual address spaces, with particular virtual addresses automatically mapped onto physical addresses for each process. Compared with segmentation, such a technique offers the illusion of contiguous physical memory and fine-grained privilege control.

To brush up the concept of virtual memory, follow along with the hand-drawn illustration in Figure 1.

Virtual Memory (x86)

On the x86, the task of mapping virtual addresses to physical addresses is managed via two tables: the *page directory* and the *page table*. Each page directory contains 1024 32-bit entries, with each entry pointing to a

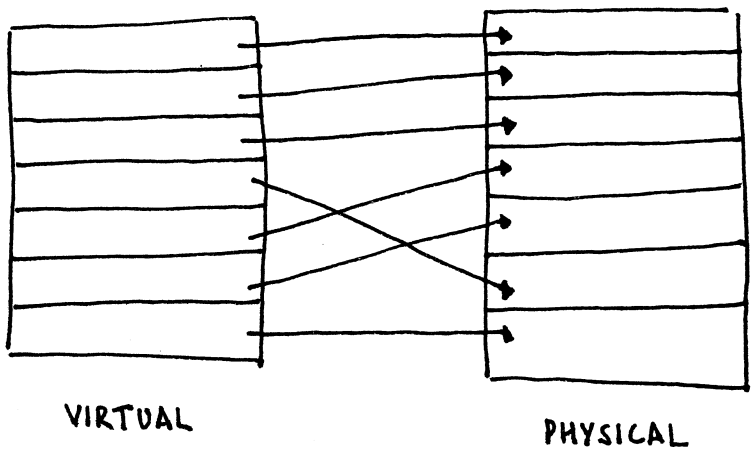


Figure 1: Virtual Memory

page table. Each page table contains 1024 32-bit entries, each pointing to a 4KiB physical frame. The page table in entirety addresses 4MiB of physical address space. The page directory, thus, in entirety addresses 4GiB of physical address space, the limit of a 32-bit address space.

The first page table pointed to by the page directory maps the first 4MiB of the virtual address space to physical addresses, the next to the next 4MiB, and so on.

The address of the page directory is loaded into a special register, the CR3.

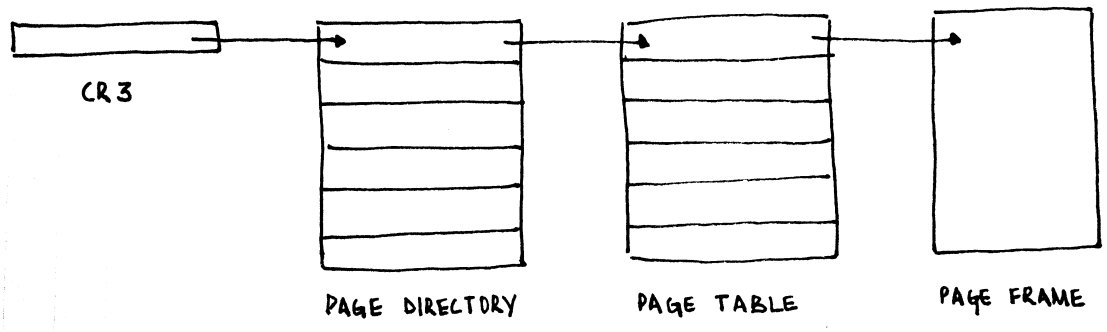


Figure 2: X86 Paging

```
; 0x8000 will be our page directory, 0x9000 will be the
; page table.

; From 0x8000, clear one 0x1000-long frame.
mov edi, 0x8000
mov cr3, edi
xor eax, eax
mov ecx, (0x1000/4)

; Store EAX - ECX numbers of time.
```

```
rep stosd

; The page table address, present, read/write.
mov dword [edi - 0x1000], 0x9000 | (1 << 0) | (1 << 1)

; Map the first 4MiB onto itself.
; Each entry is present, read/write.
or eax, (1 << 0) | (1 << 1)
.setup_pagetable:
    stosd
    add eax, 0x1000          ; Go to next physical address.
    cmp edi, 0xA000
    jb .setup_pagetable

; Enable paging.
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Extensions to the paging logic allowed 32-bit processors to access physical addresses larger than 4GiB, in the form of Physical Address Extension (PAE). The same also added a NX bit to mark pages as non-executable (and trap on instruction fetches from them).

Recursive Map

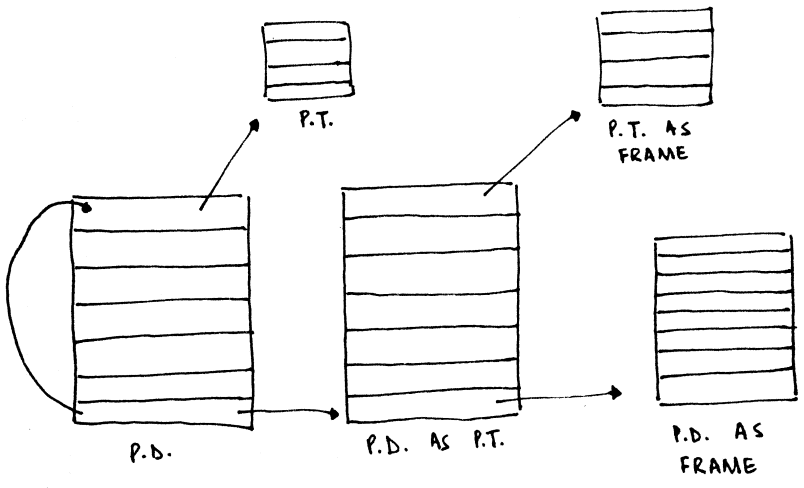


Figure 3: Recursive Page Mapping

In our simplistic case, the entire first 4 megabytes were mapped onto themselves, to so-called *identity map*. In the Real World™, however, it is often the case that the physical memory containing the page directory/tables is not mapped into the virtual address space. Instead of creating a different page table to point to the existing paging structures, a neat trick is deployed.

Before explaining the trick, note how the page directory and the page table has the exact same structure, including the attributes. What happens, then, if an entry in the page directory were to point to itself? The page directory will be interpreted as a page table. This 'page table' will have entries to actual page tables.

However, the CPU will interpret them as entries corresponding to page frames, allowing you to access them via the virtual address the page directory was self-mapped to. If that makes your head hurt, the illustration in Figure 3 might help.

Translation Lookaside Buffer (TLB)

When a virtual memory address is accessed, the CPU is required to walk through the page tables to determine the page table entry for the specified virtual address. However, walking through the page tables is slow. In the worst case, a walkthrough requires the processor to do a lookup from RAM for the page directory, followed by a lookup from RAM for the page table, where a RAM lookup latency is in the order of 100 times that of a cache lookup latency. Instead, the CPU maintains a cache of the virtual address to physical address translation, known as the Translation Lookaside Buffer (TLB).

When a virtual address is accessed, the CPU first determines if a mapping is present in the TLB. Only if the CPU fails to find one there, it walks through the actual page tables and then populates the TLB with the translation.

A problem with the TLB is that changes across the page table don't get reflected in it automatically.[4] On the x86, there exist two mechanisms to flush particular entries in the TLB:

1. The instruction 'invlpg address' invalidates the TLB entry for the page that contains 'address'.
2. Reloading CR3 with the address of a page directory flushes all the entries in the TLB. [5][6]

3.5 Till Next Time

The article got us through the backward-compatibility mess that defines the x86 boot process, into protected mode with paging enabled. In the next issue, we'll look at x86 interrupt handling, the programmable interrupt timer, multiprocessor initialization, and then the local APIC timer. We'll also answer some unanswered questions (like what happens if a page table entry doesn't exist) and conclude with a (hopefully) nifty proof-of-code.

Till then,

```
hlt:
    hlt
    jmp hlt
```

[4] *This is how PaX's PAGEEXEC emulates the NX bit by memory trapping with very little performance overhead: it sets the page table entries for the "data" pages to always trap, but allows a data access (i.e., EIP not in the accessed page) to go through. After this, it immediately resets the page table entry, but relies on the TLB for repeated page accesses to not trap. Truly, it is a work of art! –PML*

[5] CR3 is usually reloaded to change the process context (will be covered across future articles). However, a change of process does not require that the entries for the kernel pages in the TLB get flushed. To avoid so, the global bit in the page table entry can be set, and global pages can be enabled in CR4. Doing so ensures that the entry for the specific page in the TLB can only be invalidated via a 'invlpg'.

[6] The x86-64 architecture saw the introduction of tags as a part of the TLB entry, in 2008. Thus, each TLB entry is associated with a particular tag, and context switches can only involve changing of the current tag.

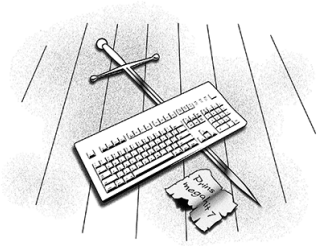
4 Prince of PoC; or, A 16-sector version of Prince of Persia for the Apple][.

by Peter Ferrie

Just in time for the 25th anniversary of Prince of Persia on the Apple][, I present to you the first ever two-sided 16-sector version!

The funny thing is that I never played it on the real Apple][, only on the PC. Even after I acquired an Apple][`.nib` version in 2009, I didn't play it. Of course, the reason for that was, I was still using ApplePC as my Apple][emulator, and it had a fatal memory-corruption bug that crashed the game. Finally in 2014, I made the switch to AppleWin. AppleWin had its own bugs, but nothing that I couldn't work around.

The retail version of the Apple][version of Prince of Persia came on two sides of a single disk. The sectors were stored in 18-sector format, and they were *full*. As a result, the 16-sector cracked versions all made use of an additional side to store those extra sectors. In 2013, about a year after the source code was recovered, Roland Gustafsson was interviewed and expressed the opinion that the three-side version "was silly and really not impressive." Taking this as a challenge, I decided to make a two-sided 16-sector version.



I started with the "rebuilt from source" version. The first thing that you will notice is that it looks different in one particular place. The reason is that whoever built it used the 3.5" settings but placed it in the 5.25" format. It means that it never asks to turn over the disk when you reach Level 3. It prompts to "insert" the disk instead, as though it is a single disk.

4.1 If you build it, they will come

So I decided to build it myself in an emulated Apple][. As no one seems to have ported Git to this platform, I went through a rather round-about ritual of converting and compiling the code.

First, I started AppleWin and formatted a DOS 3.3 disk. Onto this disk, I saved some binary files the same size as the source files, then exited AppleWin. Now that the disk was ready, I used a hex editor to change the file types to text, to avoid the need to carry the load address and size.

I converted the source code by changing all line endings from LF to CR, setting the high bit on every character and inserting them in my own tool. (I really need to port that tool to ProDOS.)

Starting AppleWin again, I used Copy][Plus to move the files from a DOS 3.3 disk to a ProDOS disk. Using the Merlin assemble, I loaded and assembled the source files, saving object files to disk. Now that the object files were ready, I copied them back to the DOS 3.3 disk with Copy][Plus and exited AppleWin.

Finally, I extracted the files with another of my own tools that needs a ProDOS port, inserted images at the appropriate locations in the track files, and used a hex editor to place those track files onto the disk image.

4.2 Try Try Again, and Again and Again

The first thing that I noticed is that it won't boot, as building the 5.25" version enabled the copy-protection, which began in the boot phase. I worked around that one by bypassing the failure check.

The second thing that I noticed is that—thanks to another layer of copy protection—you couldn't play beyond Level 2. The second-level copy protection relied on two variables, named `redherring` and `redherring2`. The `redherring` variable was set indirectly during the boot-time copy protection check. However, the

`redherring2` was never set in the source code version. Presumably someone removed the code (but did not notice that the declaration remained in the header file) because it wasn't used in the 3.5" version, because that version was not copy-protected. Unfortunately, without that value in the 5.25" version, you couldn't start the later levels. It was set in the retail 5.25" version, however, and thus we also found out that the source code was only for the 3.5" version. I bypassed this problem by writing the proper value to the proper place manually.

The third thing I noticed was that the graphics become corrupted on Level 4. The reason was yet another layer of copy-protection, which was executed before starting Level 1, but the effect was delayed until after starting Level 4. Nasty. `:-)` The end sequence was affected similarly. If the copy-protection failed, then the graphics became corrupted and the game froze on Level 14 (the reunion scene). This was an interesting design decision. If the protection was bypassed in the wrong way—by skipping the check on Level 4, instead of fixing the variable that was being compared—then that second surprise awaited. I worked around that one in the correct way, by bypassing the failure check.

The fourth thing I noticed is that the graphics became corrupted and then game crashed into text mode when starting Level 7. The reason was the final layer of copy-protection, which was executed after completing Level 1, but the effect was delayed until the start of Level 7. Very nasty. `;-)` I worked around that one by bypassing the failure check.

Finally, I checked the rest of the "rebuilt from source" version. The most important thing (depending on your point of view) was that all of the hidden parts were missing—the hidden routines (see below) and the hidden message (which was the decryption key for the original code). I also found that track `$11` was completely missing from side B, so the side B '^' routine (see below) caused a hang. Some of the graphics data were truncated, too, when compared to the retail version which I acquired in the meantime. Even though I didn't notice any difference when I played it, I gave up on that idea, and just ripped the tracks from the 5.25" retail version instead.

4.3 Turn Disk Over

Another interesting thing is how the game detects which side of the disk is in the drive. The protected version uses a unique value in the prologue data for the two sides (`$A9` and `$AD`), and uses an API to specify which one to expect. Since a standard 16-sector disk also has a standard prologue, which is identical on both sides, that was no longer an option for me. Instead, I chose to find a free sector in a location that was common to both sides, and placed the special byte there. When the prologue API was used, I redirected my read routine so that the next read request would first seek to the free sector and read the byte. If they matched, then the proper side was inserted already. Otherwise, the routine would read the sector periodically until that became true.

4.4 Size Does Matter

At a high level, the solution to the size problem is one of compression—technically, further compression, since some of the data are compressed already. However, I required a compression algorithm that packed well, was fast to decompress, and most importantly, small. The size limitation was significant. The game requires 128kb of memory, and uses almost all of it. I was fortunate enough to find a small (4096 bytes) region at `$d000` in main memory, in which to place my loader and the read buffer. This was the location of the original loader for the game. I simply replaced it with my own. I needed a read buffer within that region, because I had to load the compressed data somewhere before decompressing it into its final destination. I wanted the read buffer to be as large as possible, in order to reduce the number of read requests that I had to make. Shown in Figure 4, I managed to fit the loader code and data into under 1280 bytes: 752 bytes of code, 202 bytes for the sector table, the rest was dynamic data. That left me with 2816 bytes for the read buffer.

That space was so small that the write routine (for saving the game after you reach side B) would not fit in memory at the same time. To work around that problem, I separated the write routine, and loaded and executed it dynamically when a save request was made. It was discarded after it has done its job.

Back to the choice of compression.

I have written Apple][implementations for two well-known algorithms: LZ4 and aPLib. I did not want to write another one, so I was forced to choose between them. LZ4 was both fast and small (my implementation was only 152 bytes long), but it did not pack well enough. It had to be aPLib. aPLib packs well (about 20kb smaller than LZ4), is fast enough when factoring in the reduced number of sectors to read, and small (my implementation is only 228 bytes long, so less than one sector).

Some of the sectors are read only individually, some of them are read only as part of an entire track, and some of them are read using both methods, depending on the context. Once I determined how each of the sectors was loaded, I grouped them according to the size of the read, and then compressed the resulting block. I gave myself only two days total for the project, but it ended up taking me about two weeks. Most of that time was spent on finding an appropriate data structure.

I finally chose a variable length region set to describe the placement of the sectors within a track. This yielded a huge advantage for the sectors which were read only in track mode, when the packed size of the single region was too large for the read buffer. In that case, the file could be split into two smaller virtual regions, compressed separately to fit. The split point was determined by splitting into all 17 pairs (1 and 17, 2 and 16, 3 and 15 ...), compressing the pairs, then identifying the smallest pair. The smallest pair was chosen by the minimum number of sectors and then the minimum number of bytes. The assumption was that it costs more to decompress fewer bytes in more sectors, than to decompress more bytes in fewer sectors, even if the decompression was faster in the first case, because of the time to read and decode the additional sector. However, the flexibility of the region technique allowed the alternative case to be used without any changes to the code.

The support for the sector reads was flexible, too. Since the regions were defined only by their start and length, I could erase the individual addresses from the 18-sector requests. This allowed me to move sectors within a track, and to make the corresponding change in the 18-sector request packet. This was actually needed for track 4. For track 4, the region that began at sector `$0a` did not fit into 6 sectors even after compression. Fortunately, the region that began at sector 0 needed only 7 sectors, so the region at sector `$0a` could move to sector 9. This was enough to get it to fit. For track `$13`, the first two sectors were never accessed, so I could have moved sector 2 to sector 0, but there was no benefit to it.

Overall, my technique saved over 11 tracks on the first side, and over 16 tracks on the second side. Not enough for a single-side version, though.[7] `;-)`

4.5 And Now for Dessert: Easter Eggs!

While digging through the game code, I found several hidden routines. When playing side B, press '^' after completing a level to see an animation of Jordan waving, press a key at the end to view it again. In the byte bastards version, type RAMROD at the crack page for a hidden message.

Before booting, hold both Apple keys, then press one of the following to activate hidden modes.

DEL	Only on //GS, displays an oscilloscope.
!	Displays a message, and then a lo-res animation.
ENTER	Continually draws a fractal, press 'c' to change colors.
@	Displays a bouncing, spinning cube.
^	Pulses the drive head. Move joystick to change tone, sounds like a motorcycle.

Neighbors, is this not a tale of Shakespearean proportions and passions? A young prince, a mystery of code broken by underhanded blows in the dark, the poisoned daggers of copy-protection that even perpetrators forgot about—all laid bare by a contrived play of PoC! Is the Play the Thing, or is PoC the Thing, or are they the Thing together? You decide! –PML

[7] As a point of interest, I experimented with concatenating the entire data together, and including the sector offset in the table. That decreased the space quite significantly, but at a cost of increasing the size of the code, and making updating the data extremely difficult. That version saved over 13 tracks on the first side, and over 18 tracks on the second side. However, this was still not enough for a single-side version. In the end, it was not worth the effort, and it will not be released.

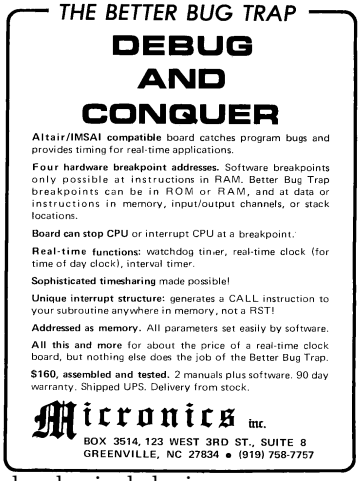
#	Side A	Side B
00		trk
01	trk	trk
02	sectors (00-0d)	trk
03	trk	trk
04	sectors (00-09, 0a-11)	sectors (00-05, 06-11)
05	trk	sectors (00-0b)
06	trk	trk
07	trk	trk
08	trk	trk
09	trk	trk
0a	trk	trk
0b	trk	sectors (00-05 / 06-11)
0c	sectors (00-05, 06-11)	sectors (00-0b / 0c-11)
0d	sectors (00-0b / 0c-11)	trk
0e	trk	trk
0f	trk	trk
10	trk	trk
11	trk	trk
12	trk	trk
13	sectors (02-11)	trk
14	sectors (04-11 / 00-03)	trk
15	trk	trk
16	trk	trk, sector 01
17	trk	sector 01
18	trk	trk
19	trk	trk
1a	trk	trk
1b	trk	sectors (00-08)
1c	trk, sectors (0d-11)	sectors (00-08 / 09-11)
1d	trk	sectors (00-08 / 09-11)
1e	trk	sectors (00-08 / 09-11)
1f	trk	sectors (00-08 / 09-11)
20	sectors (00-08, 09-11)	sectors (00-08 / 09-11)
21	sectors (00-08 / 09-11)	sectors (00-08 / 09-11)
22	sectors (02-11), trk	trk

Figure 4: Tracks and Sectors

5 A Quick Introduction to the New Facedancer Framework

by gil

Recently, I rewrote the Facedancer software stack with the goal of making it easier to write new emulators for both well-behaved and poorly-behaved devices. In this post I'm going to give an introduction to doing both. I assume you've got a Facedancer board, python3, the pyserial library, and a current revision of the code. I'll start with a very brief overview of the USB protocol itself, then show how to modify the existing USB keyboard emulator code to emulate a different (yet still well-behaved) device, and finally show how to take a well-behaved device and make it misbehave in specific ways.



5.1 USB

The USB protocol defines a bunch of abstractions: Devices, Configurations, Interfaces, and Endpoints. Some of these terms are a bit counterintuitive, understanding of which is not at all aided by how they're referred to by users.

A Device is a physical thing that gets plugged into a USB port. A single physical device may present itself to the operating system as multiple logical devices (think a keyboard with built-in trackpad or one of those annoying USB sticks that pretends it's both a USB mass storage device and a USB CD-ROM so it can install adware). In USB parlance, each of the logical devices is not a Device, but rather an Interface. I'll get to those in a couple paragraphs.

When a device is connected to a host, the host begins the enumeration process, in which it requests and the device responds with a bunch of descriptors that describe how the device can and/or wants to behave. The device presents to the host a set of "configurations"; the host chooses exactly one of these and the device, er, configures itself accordingly. But what's a configuration? It's a set of interfaces!

An Interface is a single logical device as mentioned above: a keyboard XOR a trackpad XOR an external hard drive XOR an external CD-ROM XOR... From the perspective of writing software emulators for these things, this architecture is actually kinda helpful: we can write a single interface implementing a keyboard and then include it in various device implementations. Code reuse FTW.

Each interface contains multiple "endpoints," which are the actual communication channels to and from the host. Only one endpoint is required: endpoint 0 (EP0) is the bidirectional "control" endpoint, used for exchange of descriptors on connection and optionally for asynchronous communication thereafter. (The various ways a device and host can communicate are beyond the scope of this post and, considering the tendency of device manufacturers to fabricate their own protocols to run over USB, probably intractable to cover in any single document. Your best bet to gain understanding are either fuzz it or read the device driver code.)

Endpoints other than EP0 are unidirectional so, in the case of something like an external hard drive that needs to both send and receive large amounts of data, the interface will define two endpoints: one for host-to-device ("OUT") transfers and another for device-to-host ("IN") transfers.

Lastly, the USB protocol (up to and including USB 2.0) is "speak when spoken to": all device communication is initiated by the host, which means even more state machines and callbacks than you might have been expecting.

With that, let's go to the code.

5.2 A Simple Device

All of the source files are in the "client" subdirectory of the SVN tree. You can tell the new stuff from the old:

1. The old libraries are named `GoodFET*`.
2. The old programs are named `goodfet.*`.
3. The new libraries are named `USB*` (plus `MAXUSBApp.py`, `Facedancer.py`, and `util.py`.)
4. The new programs are named `facedancer-*`.

Start by looking at `facedancer-keyboard.py`. It's pretty simple: we import some stuff, open a connection to the serial port, say we want to talk to a `Facedancer` on the serial port, then we want to talk to the `MAXUSBApp` on the `Facedancer`, and we hand this to an instance of the `USBKeyboardDevice` class, which connects the emulated device to the victim and we're off to the races. Easy enough.

The good news here is that you shouldn't have to ever worry about what goes on in the `Facedancer` and `MAXUSBApp` classes; the entirety of the logic specific to any given USB device is contained with the `USBDevice` class, of which (in this case) `USBKeyboardDevice` is a subclass. To create your own device, just create a new class that inherits from `USBDevice` and customize it as you see fit. As an example, look at `USBKeyboardDevice.py` for the implementation of the `USBKeyboardDevice` class.

Way at the bottom of `USBKeyboardDevice.py`, you'll find the definition for the `USBKeyboardDevice` class. It's fairly short: we define a single configuration (notice the configurations are numbered from 1) that contains a single interface, then we send that configuration on to the superclass initializer along with a bunch of magic numbers. These magic numbers are primarily used by the host operating system to figure out which driver to use with the attached device. From the Facedancer side, however, the keyboard functionality is implemented in the `USBKeyboardInterface` class, which takes up most of the file. Scroll back up to the top and look at that now.

The `hid_descriptor` and `report_descriptor` are hard-coded as opaque binary data specific to HID devices (I may abstract away their details at some point, but it's not a particularly high priority). In `__init__`, there's a dictionary mapping descriptor ID numbers to the actual descriptor data, which is sent to the superclass initializer (I'll get into more detail on this in the section on misbehaving devices). Also in `__init__`, a single `USBEndpoint` is instantiated, which includes a callback (`self.handle_buffer_available`).

Remember that the device never initiates a data transfer: the host will ask the device if it has any data ready; if it doesn't, the device (in our case, the MAX3420 USB chip on the Facedancer board itself) will respond with a NAK; if it *does* have data ready, the device will send the data on up. Thus whenever the host asks for data for this particular endpoint, the callback will be invoked. ("Whenever" is a bit misleading because the host will likely send polls faster than we can deal with them, but it's close enough for the time being.)

The `handle_buffer_available` method calls `type_letter`, which sends the keypress over the endpoint. (This abstraction as it stands right now is messy and is high on my list to fix—the `USBEndpoint` class should have "send" and "receive" methods, rather than having to climb up through the abstraction layers to the `send_on_endpoint` call currently in `type_letter`.)

To make a very long story short, writing an emulator for a new device should be straightforward:

1. Subclass `USBInterface` (eg, as `MyNewInterface`), define your set of endpoints and pass them to the superclass initializer, and define endpoint handler functions.
2. Subclass `USBDevice` (eg, as `MyNewDevice`), define a configuration containing `MyNewInterface`, and pass it along to the superclass initializer.

5.3 A Misbehaving Device

If you subclass `USBDevice` and `USBInterface` as described above, the rest of the class hierarchy should do the Right Thing (TM) with regards to the USB protocol itself and talking to the Facedancer to perform it: appropriate descriptors will be sent when requested by the host, correct callback functions will be called when endpoints are polled by the host, etc. But if you want to test how systems react in the face of devices that don't perform exactly as expected, you're going to have to dig in a bit.

The pattern I've tried to follow (though there are certainly deviations, which I intend to deal with—patches appreciated!) is for the `USBDevice` class to handle control messages over endpoint 0 and dispatch them to the appropriate instance of (subclasses of) `USBConfiguration`, `USBInterface`, or `USBEndpoint`. For example, if the host sends a GET_DESCRIPTOR request for the configuration, the request is dispatched to `USBConfiguration.get_descriptor`, which returns the data to be sent in response.

This logic is contained in the `USBDevice.handle_request` method; if you want your custom misbehaving device to do weird stuff for every incoming request, this is the method to override. If, on the other hand, you're looking to mess with just descriptors for a specific abstraction, you're better off overriding the `get_descriptor` method of the `USB*` classes. If you want to send non-standard responses to any of the other control messages (eg, CLEAR_FEATURE, GET_STATUS, etc), you should override the associated `handle_*_request` method of `USBDevice`. (Note that `USBDevice.handle_request` is the method that dispatched to the handle_*_request methods.)

Each of the top-level `USB*` classes (`USBDevice`, `USBConfiguration`, `USBInterface`, and `USBEndpoint`) has a `self.descriptors` member that maps from descriptor number to a descriptor or a function that returns a descriptor. Thus you are not constrained to hard-coding values, you can instead provide a function that creates whatever descriptor you want sent.

To make a somewhat less-long story short, modifying an emulated device to misbehave should be similarly straightforward.

1. Subclass whichever of `USBDevice`, `USBConfiguration`, `USBInterface`, or `USBEndpoint` contains the behavior you want to modify.
2. Override the `descriptor` dictionary in your subclass to change what descriptors get sent in response to requests.
3. Override the `handle_*_request` methods in your subclass of `USBDevice` to change how your device responds to individual requests.
4. Over the `USBDevice.handle_request` method to change how your device responds to *all* requests.

Happy fuzzing!

6 Dumping Firmware from Tamagotchi Friends by Power Glitching

by Natalie Silvanovich, Tamagotchi Merchant of Death
with the kindest of thanks to Mr. Blinky.

Figure 5: These sprites were among many dumped from the Tamagotchi Friends ROM.

The Tamagotchi Friends is the latest addition to the Tamagotchi series of virtual pet toys. Released on Boxing Day of 2013, it features NFC messaging and games as a part of a traditional Tamagotchi toy. Recently, I used glitching to dump the code of the Tamagotchi Friends.

The code for the Tamagotchi Friends is stored in mask ROM internal to its GeneralPlus GPLB series LCD controller. In the previous Tamagotchi version (the Tamatown Tama-Go), I used a vulnerability in the processing of external data from a flash accessory to dump the code, but this is not possible for the Tamagotchi Friends, as it does not support flash accessories. In fact, the Tamagotchi Friends has a substantially reduced attack surface compared to the Tamatown Tama-Go, as it also does not support infrared communications. The only available inputs on the Tamagotchi Friends are the buttons, the EEPROM (which is used to store important persistent data, like the number of slices of carrot cake your Tamagotchi has on hand) and NFC.

After eavesdropping on and simulating the NFC, and dumping and rewriting the EEPROM, I determined that they both had limited potential to contain exploitable bugs. They did both appear to fill buffers in RAM with user-controlled data in the course of normal operation though, which meant they both could be useful for creating shellcode buffers in the case that there was a bug that allowed the program counter to be moved to the buffer.

One possible way to move the program counter was glitching, basically driving unexpected signals into the microcontroller and hoping that they would somehow cause that program counter to change and by chance land in the shell code buffer. Considering that memory space of the microcontroller is 65,536 bytes, and the largest buffer I could fill with a NOP slide is roughly 60 non-contiguous bytes this sounds like a long shot, but the 6502 architecture used by the microcontroller has some properties that makes random program counter corruption more likely to lead to code execution compared to other architectures. To start, it has no memory validation, so any access of any address will succeed, regardless of whether any memory is mapped to the location. This means that execution will not stop even if an invalid address is accessed. Also, invalid

opcodes on 6502 are guaranteed to execute in a finite amount of time[8] with undefined behaviour, so they also will not stop execution. Together, these properties make it very unlikely that execution will ever stop on a 6502 processor, giving shellcode a lot of chances to get executed in the case that the program counter is corrupted.

Another useful feature of this particular microcontroller is that the RAM starts at address zero, and the lowest hundred bytes or so of RAM is used by the SPU and is often zero. In 6502, zero is the opcode for BRK, which acts like NOP if a debugger is not attached, so this RAM could potentially act as a NOP slide. In addition, in the Tamatown Tama-Go (and I assumed the Tamagotchi Friends), the EEPROM is copied to address `0x300`, which is still fairly low in RAM addresses. So if the program counter got set to zero, there is a possibility it could slide through RAM up to the EEPROM. Of course, not every value in RAM before `0x300` is zero, but if enough are, it is likely that the other values will be interpreted as instructions that don't alter the program counter's course some portion of the time.

Since setting the program counter to zero seemed especially likely to cause code execution, I started by glitching the input power, as this had the potential to clear the program counter. The Tamagotchi Friends has three types of volatile memory: registers like the program counter, DPRAM (used for the LCD) and SRAM. DPRAM and SRAM both have fairly long persistence after they stop being powered, so I hoped if I cut the power to the microcontroller for a short period of time, it would corrupt the registers, but not the RAM, and resume execution with the program counter at address zero.

I tried this using an Arduino to switch the power on and off at different speeds. For very fast speeds, the Tamagotchi didn't react at all, and for very slow speeds, it would reset every cycle. I eventually settled on cycling every five milliseconds, which had a visible erratic impact on the Tamagotchi after each cycle. At this rate, the toy was displaying an unexpected image on the LCD, corrupting the LCD, playing Yankee Doodle or screeching loudly.

I filled up the EEPROM with a large NOP slide and some code that caused a write to the LCD screen, reset the Tamagotchi so the EEPROM was downloaded into RAM, and cycled the power. Roughly one out of every ten times, the code executed and wrote the LCD.

I then moved the code around to figure out the size of the available code buffer. Two things limited the size. One is that only a small part of the EEPROM is copied into RAM at once, and the rest is only loaded if needed. The second is that some EEPROM addresses are validated. For some of these addresses, containing very critical values, the EEPROM is wiped immediately if the Tamagotchi detects an invalid value. These addresses couldn't be used for code at all. Some other less critical values get overwritten if they are invalid. For example, if a Tamagotchi is a child, but is married, the "is married" flag will be reset to the correct value. These addresses could be changed, but there was no guarantee they would stay the correct value, so I ended up jumping over them. This left exactly 54 bytes for code. It was tight, but I was able to write code that dumped the ROM over SPI through the Tamagotchi buttons in that space

The following is the shellcode I used:

```
SEI ; disable the low battery interrupt
LDA #$FF
```

[8] A few people have mentioned to me that there are certain versions 6502 processors for which this is not true, but this is definitely the case for GeneralPlus controllers.

```
STA $3011 ; port direction
STA $1109 ; LCD indicator
STA $00C5
STA $00C6
LDX #$08
LDA ($C5),Y ; No room to initialize Y.  Worst case,
ASL A       ; it will be set to 0 at the end of the loop.
LDY #$01
BCC $001A
LDY #$03
BNE $0020 ; These 4 bytes get altered before execution.  Jump over them.
NOP
NOP
NOP
NOP
NOP
STY $3012
LDY #$00
STY $3012
DEX
BNE $0013
INC $00C5
BNE $000F
INC $00C6
BNE $000F
LDA #$00
STA $3000
BNE $000F ; Branches are shorter than jumps, so use implied conditions.
```

In memory, this shellcode is as follows:

```
300: 32 17 02 01 02 01 09 00 1A 00 1A 1A 1A 1A 1A 1A
310: 20 FF 06 10 01 FF FF 02 77 77 77 77 77 77 77 77
320: 77 77 77 77 77 05 04 FF 77 77 55 00 77 77 7F 00
330: FF FF 40 EA EA EA EA EA 00 00 00 00 00 00 00 00
340: 03 78 A9 FF 8D 11 30 8D 09 11 8D C5 00 8D C6 00
350: A2 08 B1 C5 0A A0 01 90 02 A0 03 D0 04 EA 00 00
360: 03 EA 8C 12 30 A0 00 8C 12 30 CA D0 E7 EE C5 00
370: D0 DE EE C6 00 D0 D9 4C 4B 03 15 11 4C 38 00 00
```

The code begins at `341` and ends at `376`, which are the bounds of the buffer copied from the EEPROM. The surrounding values are typical values of the surrounding RAM which are not consistent across each time code is executed. The `0x03` before the beginning of the code is written after the buffer, and is an undefined instruction in 6502. Unfortunately, this means that there isn't room for any NOP sled, the program counter needs to end up at exactly the right address.

One useful feature of this shellcode is that the first seven instructions aren't strictly necessary! The registers are often the right value, or an acceptable value by chance, which gives the program counter a bit more leeway in the case that it jumps a bit beyond the beginning of the code.

I dumped all thirty-two pages of ROM using this shellcode, and they appear to be accurate. Figure 5 shows the highlights of the dump, organized by cuteness in descending order.

7 Lenticrypt: a Provably Plausibly Deniable Cryptosystem; or, This Picture of Cats is Also a Picture of Dogs

by Evan Sultanik

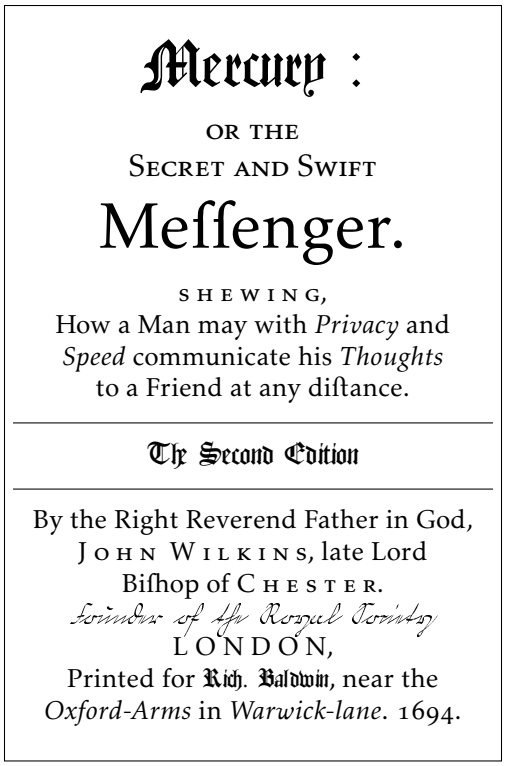
Deniable cryptosystems allow their users to plausibly deny the existence of the plaintext content of their encrypted data. There are many existing technologies for accomplishing this (*e.g.*, TrueCrypt), which usually accomplish it by having multiple separate encrypted volumes in the ciphertext that will decrypt to different plaintexts depending on which decryption key is used. Key k_1 will decrypt to innocuous volume v_1 whereas key k_2 will decrypt to high-value volume v_2. If an adversary forces you to reveal your secret key, you can simply reveal k_1 which will decrypt to v_1: the innocuous volume full of back-issues of PoC||GTFO and pictures of cats. On the other hand, if the adversary somehow detects the existence of the high-value volume v_2 and furthermore gains access to its plaintext, the jig is up and you can no longer plausibly deny its contents' existence. This is a serious limitation, since the high-value plaintext might be incriminating.

An *ideal* deniable cryptosystem would allow the creator of the ciphertext to plausibly deny having created the plaintext *regardless* of whether the true high-value plaintext is revealed. The obvious use-case is for transmitting illegal content: Alice wants to encrypt and send her neighbor Bob a pirated copy of the ColecoVision game *George Plimpton's Video Falconry*. She doesn't much care if the plaintext is revealed, however, she *does* want to have a plausible *legal* argument in the event that she is prosecuted whereby she can deny having sent that particular file, *even if* the high-value file is revealed. In the case of systems like TrueCrypt, she can't really deny having created the alternate hidden volume containing the video game since the odds of it just randomly occurring there *and* a key happening to be able to decrypt it are astronomically small. But what if, using our supposed "ideal" cryptosystem, she *could* plausibly claim that the existence of the video game was due to pure random chance? It turns out that's possible, and we have the PoC to prove it!

Before we get to the details, let's first dispel the apparent nefariousness of this concept by discussing some more legitimate use-cases. For example, we could encrypt a high-value document such that it decrypts to either a redacted or unredacted version depending on the key. If the recipients are not aware that they have unique keys, one could deliver what *appears* to be a single encrypted message to multiple recipients with individualized content. The individualization of the content could also be very subtle, allowing it to be used as a unique watermark to identify the original source of a leaked document: a so-called "canary trap." Finally, "deep-inspection" filters could be evaded by encrypting an innocuous payload with a common, guessable password.

7.1 Running Key Ciphers

A running key cipher is one of the most basic cryptosystems, yet, if used properly, it can be one of the most secure. Being avid PoC||GTFO readers, Alice and Bob both have a penchant for treatises with needlessly verbose titles that are edited by Right Reverend Doctors. Therefore, for their secret key they choose to use a copy of a seminal work on cryptography by the Rt. Revd. Dr. Lord Bishop John Wilkins FRS.


Mercury :

OR THE

SECRET AND SWIFT

Meſſenger.

SHEWING,

How a Man may with *Privacy* and
Speed communicate his *Thoughts*
to a Friend at any diſtance.

The Second Edition

By the Right Reverend Father in God,
JOHN WILKINS, late Lord
Biſhop of CHESTER.
Founder of the Royal Society
LONDON,
Printed for Rich. Baldwin, near the
Oxford-Arms in *Warwick-lane*. 1694.

They have agreed to start their running key on the first line of the book, which reads:

> “Every rational creature, being of an imperfect and dependant Happineſs, if therefore naturally endowed with an Ability to communicate its own Thoughts and Intentionſ ; that ſo by mutual Services, it might better promote it ſelf in the Proſecution of its own Well-being.”

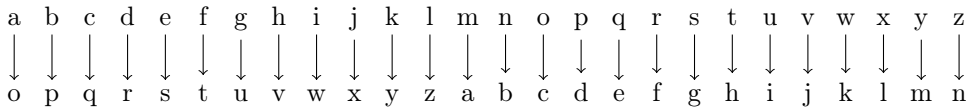
The encryption algorithm is then very simple: Each character from the running key is used as a rotation to permute the associated character of the plaintext. For example, say that the first character of our plaintext is “A”; we would take the first character of our running key, “E”, look up its numerical index in the alphabet, and rotate the plaintext by that much to produce the ciphertext.

```
PLAINTEXT:   AN ADDRESS TO THE SECRET SOCIETY OF POC OR GTFO...
RUNNING KEY: EV ERYRATI ON ALC REATUR EBEINGO FA NIM PE RFEC...
CIPHERTEXT:  EI EUBIELA HB TSG JICKYK WPGQRZM TF CWO DV XYJQ...
```

There are of course many other ways the plaintext could be combined with the running key, another common choice being XORing the bits. If the running key is truly random then the result will almost always be what is called a “one-time pad” and will have perfect secrecy. Of course, my expository example is nowhere near secure since I preserved whitespace and used a running key that is nowhere near random. But, in practice, this type of cryptosystem can be made very secure if implemented properly.

7.2 Book Ciphers

Perhaps the *most* basic type of cryptosystem—one that we’ve all likely independently discovered in our early childhood—is the substitution cipher: Each letter in the alphabet is statically mapped to another. The most common substitution cipher is ROT13, in which the letters of the alphabet are rotated 13 steps.



In fact, we can think of the running key cipher we described above as a sort of substitution cipher in which the alphabet mapping changes for each byte based off of the key.

Book Ciphers marry some of the ideas of substitution ciphers and running key ciphers. First, Alice and Bob decide on a shared secret, much like the book they chose as a running key above. The shared secret needs to have enough entropy in order to have at least one instance of every possible byte in the plaintext. For each byte in the shared secret, they create a lookup table mapping all 256 possible bytes to lists containing all indexes (*i.e.*, file offsets) of the occurrences of that byte in the secret:

```
with open(secret_key_file) as s:
    indexes = dict([(b, []) for b in range(256)])
    for i, b in enumerate(map(ord,s.read())):
        indexes[b].append(i)
```

Then, for each byte encountered in the plaintext, the ciphertext is simply the index of an equivalent byte in the secret key:

```
def encrypt(plaintext, indexes):
    for b in map(ord, plaintext):
        print random.choice(indexes[b]),
```

To decrypt the ciphertext, we simply look up the byte at the specified index in the secret key:

```
def decrypt(ciphertext, secret_key_file):
    with open(secret_key_file) as s:
        for index in map(int, ciphertext.split()):
            s.seek(index)
            sys.stdout.write(s.read(1))
```

In effect, what is happening is that Alice opens her book (the secret key), finds indexes of characters that match the characters she has in her plaintext, writes those indexes down as her ciphertext, and sends it to Bob. When Bob receives the ciphertext, he opens up his identical copy of the book, and for each index he simply looks up the letter in the book and writes that down the letter into the decrypted plaintext. There are various optimizations that can be made, *vi&.*, using variable-length codes within the key similar to LZ77 compression (*e.g.*, using words from the book instead of individual characters).

7.3 Lenticular Book Ciphers

In the previous section, I showed how a book cipher can be used to encrypt plaintext p_1 to ciphertext c using secret key k_1. In order for this to be useful as a plausibly deniable cryptosystem, we will need to ensure that given some *other* secret key k_2, the *same* ciphertext c will decrypt to a totally different plaintext p_2. In this section I'll discuss an extension to the book cipher which achieves just that. I call it a "Lenticular Book Cipher," inspired by the optical device that can present different images to the viewer depending on the lens that is used. I was unable to find any description of this type of cryptosystem in the literature, likely because it is very naïve and practically useless ... except for in the context of our specific motivating scenarios!

Given a set of plaintexts $P = \{p_1, p_2, \ldots, p_n\}$ and a set of keys $K = \{k_1, k_2, \ldots, k_n\}$, we want to find a ciphertext c such that `decrypt(`c`,`k_i`)` $\mapsto p_i$ for all i from 1 to n. To accomplish this, let's consider an individual byte within each of the plaintexts in P. Let $p_i[j]$ represent the j^{th} byte of plaintext i. Similarly, let's define $k_i[j]$ and $c[j]$ to refer to the j^{th} byte of a key or the ciphertext. In order to encrypt the first byte

of all of the plaintexts, we need to find an index m such that $k_i[m] = p_i[0]$ for i from 1 to n. In general, $c[\ell]$ can be any unsigned integer m such that

$$\forall i \in 1, \ldots, n : k_i[m] = p_i[\ell].$$

We can relatively efficiently find such an m by modifying the way we build the `indexes` lookup table:

```
def build_index(secret_keys):
    indexes = {}
    for i, key_bytes in enumerate(zip(*secret_keys)):
        key_bytes = tuple(map(ord, key_bytes))
        if key_bytes not in indexes:
            indexes[key_bytes] = [i]
        else:
            indexes[key_bytes].append(i)
    return indexes
```

Encryption then happens similarly to the regular book ciper:

```
def encrypt(plaintexts, secret_keys):
    indexes = build_index(secret_keys)
    for text_bytes in zip(*plaintexts):
        text_bytes = tuple(map(ord, text_bytes))
        print random.choice(indexes[text_bytes]),
```

Decryption is identical to the regular book cipher.

So, in fewer than twenty lines of Python, we have coded a PoC of a cryptosystem that allows us to do the following:

```
encrypt([open("plaintext1").read(), open("plaintext2").read()],
        [open("key1").read(), open("key2").read()])
```

If we pipe STDOUT to the file "`cipher.enc`", we can decrypt it as follows:

```
with open("cipher.enc") as enc:
    decrypt(enc.read(), "key1") # This will print out plaintext1
    decrypt(enc.read(), "key2") # This will print out plaintext2
```

There do seem to be a number of limitations to this cryptosystem, though. For example, what keys should Alice use? The keys need to be long enough such that every possible combination of bytes that appears across the plaintexts will occur in `indexes`; the length of the keys will need to increase exponentially with respect to the number of plaintexts being encrypted. Fortunately, in practice, you're not likely to ever need to encrypt more than a few plaintexts into a single ciphertext. One possible source of publicly available keys to use would be YouTube videos: Alice could simply download a video and use its raw byte stream as the key. Then all she needs to do is communicate the name of or link to the video to Bill off-the-record.

I have created a complete and functional implementation of this cryptosystem, including some optimizations (*e.g.*, variable block length, compression, length checksums, error checking, *&c.*). It is available here:

`https://github.com/ESultanik/lenticrypt`

7.4 Proving a Cat is Always Also a Dog

So far, I've gone through a lot of trouble to describe a cryptosystem of dubious information security[9] whose apparent functionality is already available from tools like TrueCrypt. In this section I will make a

[9]While I do have a few letters after my name that suggest I know a thing or two about Computer Science, cryptography is not my specific area of specialization.

mathematical argument that provides what I believe to be a legal basis for the plausible deniability provided by lenticular book ciphers, enabling its use in our motivating scenarios.

Laws and contracts aren't interpreted like computer programs; legal decisions are often dictated less by the defendant's actions than by his or her *intent*. In other words, if it appears that Alice *intended* to send Bob a copy of Video Falconry, she will be found guilty of piracy, regardless of how she conveyed the software.

But what if Alice legitimately only knew that key k_1 decrypted c to a picture of cats, and didn't know of its nefarious use to produce a copy of Video Falconry from k_2? How likely would it be for k_2 to produce Video Falconry simply by coincidence?

For sake of this analysis, let's assume that the keys are documents written in English. For example, books from Project Gutenberg could be used as keys. I am also going to assume that each character in a document is an independent random variable. This is a rather unrealistic assumption, but we shall see that the asymptotic properties of the problem make the issue moot. (This assumption could be relaxed by instead applying Lovász's local lemma[10].)

First, let's tackle the problem of figuring out the probability that `decrypt(`c`,`k_2`)` $\mapsto p_2$ completely by chance. Let n be the length of the documents in characters and let $m < n$ be the minimum required length of a string for that text to be considered a copyright violation (*i.e.*, outside of fair use). The probability that `decrypt(`c`,`k_2`)` contains no substrings of length at least m from p_2 is

$$(1 - q^m)^{(n-m+1)},$$

where q is the probability that a pair of characters is equal. Here we have to take into account letter frequency in English. Using a table from Wikipedia[11], I calculate q to be roughly 6.5 percent (it's the sum of squares of the values in the table). According to Google, there are about 130 million books that have ever been written[12]. Let's be conservative and say that two million of them are in English. Therefore, the probability that *at least one pair* of those books will produce a copyrighted passage from c is

$$1 - \left((1 - q^m)^{(n-m+1)}\right)^{\binom{2000000}{2}},$$

which is extremely close to 100% for all $m < n \ll 2000000$.

Therefore, for any ciphertext c produced by a lenticular book cipher, it is almost certain that there exists a pair of books one can choose that will cause a copyright violation! Even though we don't know what those books might be, they must exist!

Proving that this is a valid *legal* argument—one that would hold up in a court of law—is left as an exercise to the reader.

[10] Paul Erdős and László Lovász. *Problems and results on 3-chromatic hypergraphs and some related questions.* Infinite and finite sets (Colloq., Keszthely, 1973; dedicated to Paul Erdős on his 60th birthday), Volume II, North-Holland, Amsterdam, 1975, pp. 609–627. Colloq. Math. Soc. János Bolyai, Volume 10.

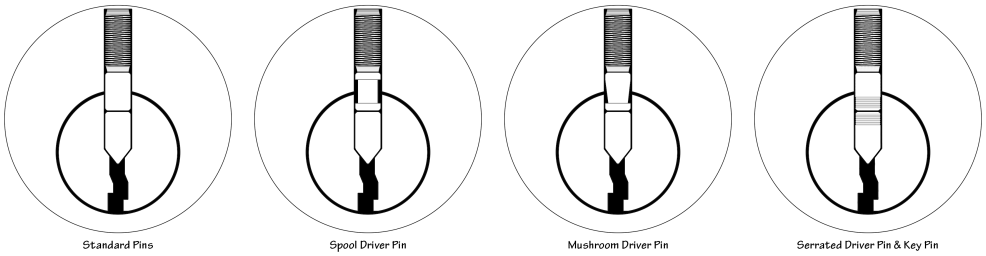
[11] `http://en.wikipedia.org/wiki/Letter_frequency#Relative_frequencies_of_letters_in_the_English_language`

[12] Leonid Taycher. *Books of the world, stand up and be counted! All 129,864,880 of you.* August 5, 2010. `http://booksearch.blogspot.com/2010/08/books-of-world-stand-up-and-be-counted.html` Retrieved March 21, 2014.

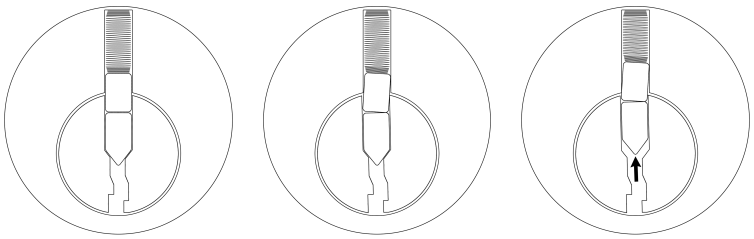
8 Hardening Pin Tumbler Locks against Myriad Attacks for Less Than a Sawbuck

by Deviant Ollam, Merchant of Dead Locks

In 1983, the renowned locksmith and physical security icon Gerry Finch submitted a brief article to Keynotes magazine, a publication of the Associated Locksmiths of America. In it, he described why it was his belief that serrated pins within a lock were superior to spool pins, mushroom pins, or any other kind of manipulation-resistant pins commonly-used in locks. Despite being very popular and well-received at the time, such wisdom appears to have faded away somewhat among locksmithing circles. This article is a re-telling of Finch's original advice with updated diagrams and images, in the hopes that folk might realize that some of the old ways are often still some of the best ways of doing things.

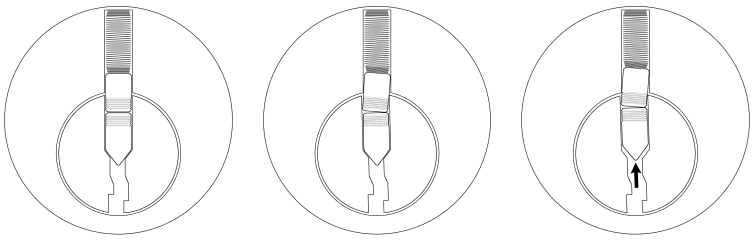


Pick-resistant pins are designed to interfere with the most common methods of attacking pin tumbler locks. Conventional operation of a lock involves first pushing the pin stacks to their appropriate positions and then turning the plug. Lockpicking, however, is performed by first applying turning pressure to the plug, then—subsequent to that—the pushing of the pins stacks is performed, with pick tools instead of a key. The following images document this process.



Pick-resistant pins make such an attack difficult by interfering with the easy movement of pin stacks if a lock's plug is already subject to turning pressure. While standard operation of the lock is still possible (in the absence of any turning pressure, the blade of a user's key will still push the pin stacks smoothly) attempts to turn, then lift (which is how picking is performed) become much more complicated. If inclined, one may acquire entire pinning kits consisting of such special pins from locksmiths supply companies. Seen in the photo below is the tray of an "S-pin" security kit from LAB.

The following images show how the ridges of a serrated pin make for additional friction during a typical lock-picking attack.



While other styles of pick-resistant pins are available on the market (such as the spool style or mushroom style seen in an earlier diagram) it was the serrated style which captured Gerry Finch's attention and became his favorite means of bolstering a lock's ability to resist attack. Part of his reason pertained to the fact that the ridges on a serrated pin are far less pronounced than on a spool or mushroom style pin. When performing the picking process, a skilled attacker can often discern quite clearly the moment when they have encountered a spool or mushroom driver pin. Due to the large ridge present and the very noticeable way in which a lock's plug will tend to turn (but the lock will fail to open) this information leakage will offer up valuable insight to an attacker. Serrated pins give away far less detail to someone who is using lockpicks.

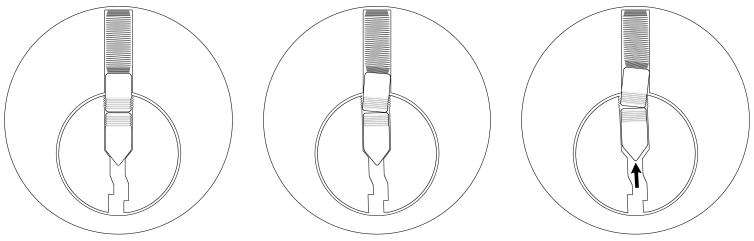
The very small ridges found on serrated pins also lend themselves to another, more substantial, means of preventing attacks against pin tumbler locks, however. Although it was not common practice at the time, Gerry Finch proposed something in the early 1980s which dazzled the locksmith community. Specifically, he advocated the process of using a thin thread-tapping tool to create additional ridges inside of a lock's plug, within the chambers where the pins are installed.

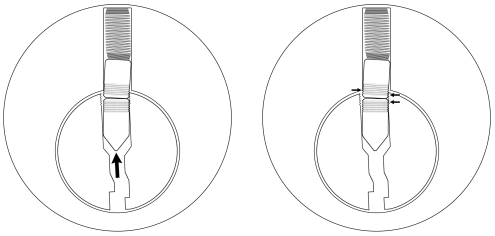
By cutting these threads into the pin chambers, a much greater degree of friction and positive lock-up between the pins and the plug can be achieved. If there is turning pressure on the plug–as there is with a lockpicking attack—and any attempt to push the pin stacks is made, the serrations will bite together. This is remarkably robust for a number of reasons:

- Even if a dedicated lockpicker gets past one region of friction, serrated edges offer repeated additional blockades to progress. Spool pins or mushroom pins typically offer only one point of resistance in each pin stack.

- The positive lock-up between pins and the plug is achieved by the driver pins and also by the key pins (if serrated key pins are installed) and for this reason this style of configuration should also offer some resistance to impressioning attacks, as well.

The following images show the mechanism by which serrated pins and thread-tapped plug chambers work in concert to resist picking attacks.





It is those particular points indicated by the small arrows where the ridges and threading jam together tightly. NOTE—As seen in the earlier photo of the field-stripped plug, I did not opt to run a tap through *all* of the pin chambers. The front-most chamber was left plain and no serrated pins would be installed there. This not only conceals the presence of such pins in the lock (at least from cursory inspection) but it affords one the opportunity to install hardened anti-drill pins in that front chamber.

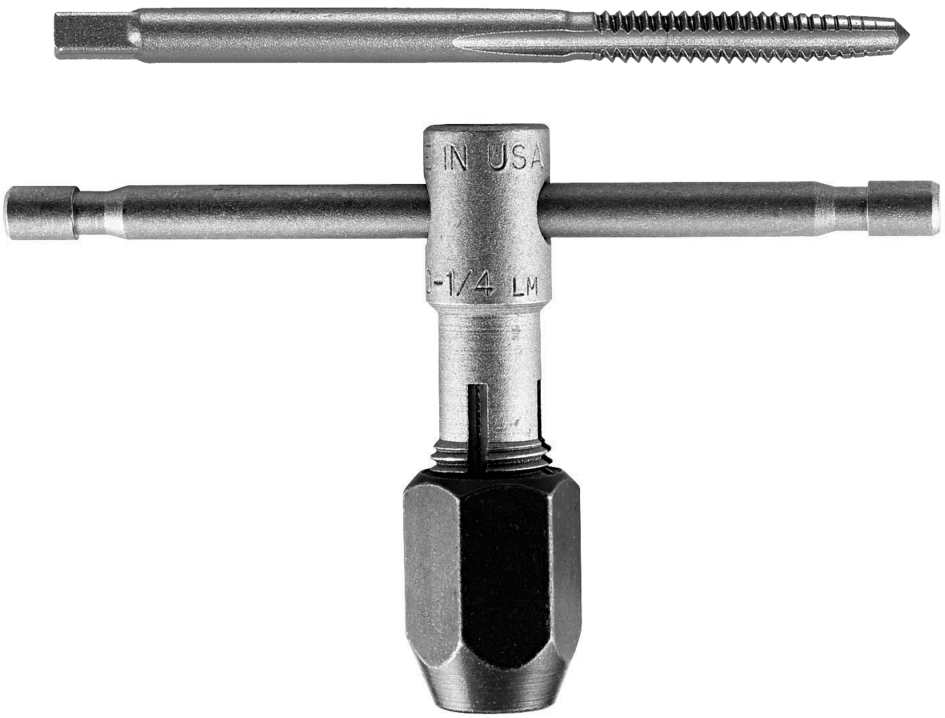
Gerry Finch suggested that course of action, as well. He also cautioned locksmiths against working a tapping tool too deeply in each chamber. He recommends a maximum of three turns per chamber, no more.

Finch's ideas proved so effective, and locks prepared in this manner tend to be so resistant to against even dedicated attacks, that the LAB company started including a 6/32" tap in some of their S-pin kits. But perhaps a little surprisingly, after all these years the practice has become so uncommon that few locksmiths with whom I have spoken nowadays even know what the tap tool is for.

If you have the knowledge of even basic lock field-stripping, it is quite possible to upgrade a pin tumbler lock using this technique for very little cost. The LAB company's S-pins are available for less than a dime each[13] and hardware tool suppliers sell both the 6/32" tap and a suitable tap handle for four dollars apiece.

Best of luck upgrading your security if you try this yourself. With a little care and dedication and for less than one Hamilton you could make your locks a great deal more resistant to attacks by someone like me.

[13]While this is technically true, such pins are commonly sold in packages of 100. So you're often out six to seven dollars for the bag, and a variety of sizes of key pins and driver pins are needed to do the job properly. It's best to find a friendly locksmith who might sell you a handful of individual pins for a few dollars.



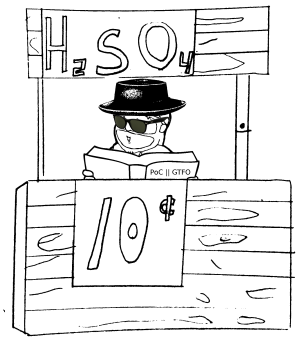
Gerry Finch was a legend in the lockpicking and locksmithing community, developing tools, techniques, instructional courses, and published works throughout his career. A veteran of the US Air Force (ret 1964) he also worked with the US Army Technical Intelligence Center teaching their Defense Against Methods of Entry course. Finch is the recipient of the Locksmith Ledger's Hall of Fame Award, The California Locksmith Association's Golden Key Award, Associated Locksmiths of America's President's Award, the Lee Rognon Award, the Gerald Connelly Pioneer Award, and the Philadelphia Award. He retired officially in 1996, but I still wouldn't want to go head-to-head with him in a picking contest.

9 Introduction to Reflux Decapsulation and Chip Photography

by Travis Goodspeed

Howdy y'all,

Unlike my prior articles for PoC||GTFO, this one is an introductory tutorial. If you are already stripping and photographing microchips, then there will be little for you to learn here. If, however, you want to photograph a chip and don't know where to begin, this is the article for you.



I'm also required by my own conscience and by good taste to warn you that if you attempt to follow these instructions, you will probably get a little bit hurt. Please be *very fucking careful* to ensure that you only get a little bit hurt. If you have any good sense at all, you will do this in a proper chemistry lab with the assistance of professionals rather than rely on this hobbyist guide. If you don't know whether to add water to acid or acid to water, and why you will hurt yourself *a lot* if you don't know, please stop reading now and take a community college course with a decent lab component.

9.1 Chemistry Equipment

At a bare minimum, you will need high-strength nitric acid (HNO_3) and sulfuric acid (H_2SO_4). Laws for acquiring these vary by country, and if you're in a jurisdiction that cares too much about the environment, you might need to use a different method.[14] In addition to the two acids, you will need isopropyl alcohol and acetone as solvents for cleaning.

Beyond the chemicals, you will need a bit of glassware. Luckily, the procedure is simple enough, so some test-tubes, beakers, and a ring stand with buret clamps will do. If you get second-hand clamps, be aware that metal should not directly touch the glass of the test tube; your clamp might be missing a rubber or cloth piece to prevent scratches.

The acids that you are working with can attack metals, so get several acid-resistant tweezers. I learned a while ago that tweezers get lost or bent, so buy a dozen and you won't have to worry about it again.

Because the acid fumes, particularly the nitric acid fumes, are so noxious, you will need a fume hood to properly contain the acid gas that boils out of the test tube when you screw up the heat.

As a handy indicator of where the acid fumes are going, I save thermal paper cardstock from air and rail tickets. They turn red or black in the presence of acid fumes, and by balancing one above the test tube I get a visual warning that the fumes have spread too far.

You could get by with a toothbrush and solvent for cleaning the chip surface, but an ultrasonic bath with solvent is better. Cheap ultrasonic cleaners are available for cleaning jewelry, and they work well enough, but be careful not to let your cleaning solvents dissolve their exposed plastic.

Finally, you will need a source of regulated heat. At this point, you're probably itching to strike off a Bunsen burner, but those are really a terrible choice. Instead, I use a cheap SMD rework soldering station, the Aoyue 850A. By turning the airflow near maximum and slowly raising the temperature, I can heat the test tube to a consistent temperature.

9.2 Chemistry Procedure

Your sample should be the smallest package of the target chip you can purchase. For a specific example, the Texas Instruments MSP430F2012 is available as PDIP (Plastic Dual Inline Package) and QFN (Quad Flat No-leads) among other packagings. While this procedure works for either, the QFN package is much smaller and has less plastic to be etched away, so it will consume far less of your nitric acid.

Begin by connecting the buret clamp to your ring stand as shown in Figure 6, with the SMD rework station's wand held just beneath the bottom of where the test-tube will be. Do not turn on the heat yet.

[14] I've heard that the Germans get good results with kolophonium, better known as rosin.

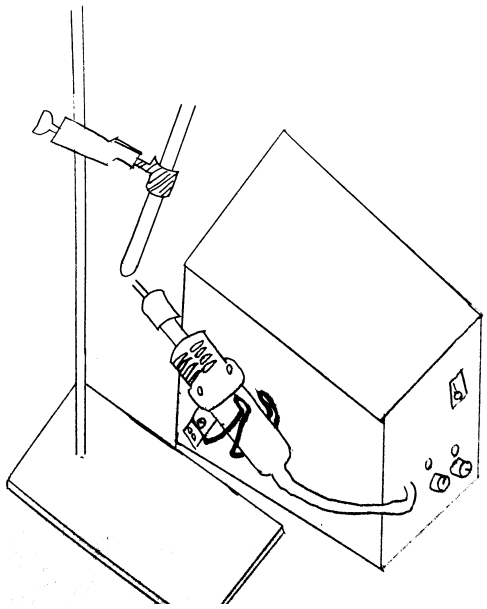
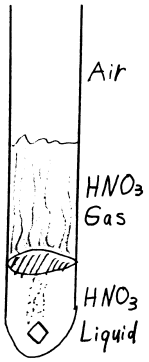


Figure 6: The clamp stand holds the test-tube next to the SMD rework station.

Place the chip into the test-tube with enough nitric acid to cover the chip and optionally add just a splash of sulfuric acid to make it attack the plastic instead of the bonding wires. For safety reasons, you will very quickly learn to do this while the glass is cold, just as you will very quickly and rather painfully learn that cold glass looks exactly like hot glass.

Place the test tube into the buret clamp. The tube should be slightly tilted, with the bottom closer to you than the top so that any explosive eruptions of boiling acid go away from your face.

With the chip covered in acid, turn the SMD rework station on with high speed and low heat. Slowly raise the temperature while watching the well-lit column of the test tube. The idea here is to create a poor man's reflux, in which the acid boils but the column of acid vapor above it remains beneath the lid of the test tube, unable to spill out. Shining a laser pointer into the tube will reveal the exact height of the column, as the laser is scattered by the acid but not by clean air.



Overheating the test tube will cause the acid to steam out, filling either the fume hood or your lab with acid fumes. All of the iron in the room will rust, your lungs will burn, and the fire alarm will trigger. Don't do this.

As the chip boils in nitric acid, the packaging will crumble off in chunks. This crumbling should continue until either the chip's die is exposed or the acid is spent.

You might notice the acid solution changing color. HNO_3 turns green or blue after dissolving copper, which greatly reduces its ability to break apart the plastic. Once the acid is spent, let the test-tube cool and then spill its contents into a beaker.

At this point, the acid might not be strong enough to further break apart the packaging, but it's still strong enough to burn your skin. HNO_3 burns don't hurt much at first, and light ones might go unnoticed except for a yellowing of the skin that takes a week or so to peel off. Sometimes you'll notice them first as an itch, rather than a burn, so run like hell to the sink if a spot on your hand starts itching. H_2SO_4 burns more like you'd expect from Batman cartoons, with a sharp stinging pain. It results in a red rash instead of

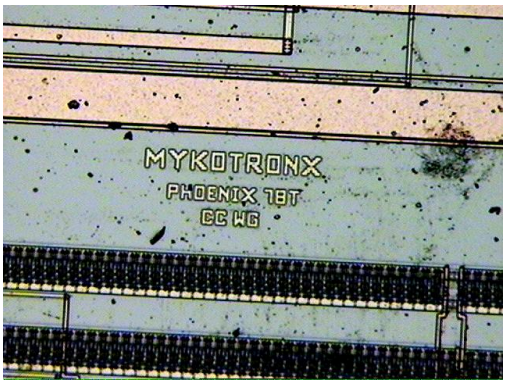


Figure 7: This is one photo of 1,475 that I took of the Clipper Chip.

yellowed skin.[15]

So now that you know better than to stick your fingers into the beaker of acid, use tweezers to carefully lift the die out of the acid and drop it into a second beaker of acetone. This beaker–the acetone beaker–goes into the ultrasonic bath for a few minutes. At this point the die will be partially exposed with a bit of gunk remaining, but sometimes larger chips will still be covered.

For best quality, the HNO_3 should be repeated until very little of the gunk is left, then a bath of only H_2SO_4 will clean off the last bits before photography.

These two acids are very different chemicals, and you will find that the H_2SO_4 bath behaves nothing like the HNO_3 baths you've previously given the chip. H_2SO_4 has a much higher boiling point than HNO_3, but it's also effective against the chip packaging well beneath its boiling point. You will also see that instead of flaking off the packaging, H_2SO_4 dissolves it, taking on an ink-black color through which you won't be able to see the sample.

After the final H_2SO_4 bath, give the chip one last trip through the ultrasonic cleaner and then it will be ready to photograph.

9.3 Photographic Equipment

Now that you've got an exposed die, it's time to photograph it. For this you will need a metallurgical microscope, meaning one that gives an image by reflected rather than transmitted light.

Microscope slides work for samples, but they aren't really necessary, because no light comes up from the bottom of a metallurgical microscope anyways. Small sample boxes with a sticky surface are handier, as they are less likely to be damaged in a fall than a case full of glass microscope slides.

For photographing your chip, you can either get a microscope camera or an adapter for a DSLR. Each of these has its advantages, but the microscope cameras are very often just cheap webcams with awkward Windows-only software, so I go the DSLR route. Through either sort of camera, you can take individual photos like the one in Figure 7.

[15]Here's a handy rhyme to remember safety:

> *Johnny was a Chemist's Son,*
> *but Johnny is No More.*
> *What Johnny thought was* H_2O*,*
> *was* H_2SO_4*!*

9.4 Photographic Procedure

Whichever sort of camera you use, you won't be able to fit the entire chip into your field of view. In order to get an image of the whole chip, you must first photograph it piecemeal, then stitch those photos together with panorama software.[16]

Begin at a known corner of the chip and take a series of photographs while moving in the same direction and keeping the top layer of your sample in focus. Each photograph should overlap by roughly a third its contents with the image before and after it, as well as those on adjacent rows. Once a row has been completed, move on to the next row and move back in the opposite direction.

Once you have a complete set of photos, load them in Hugin on a machine with plenty of RAM. Hugin is a GUI frontend to panorama utilities, and it allows you to correct mistakes made by those tools if there aren't too many of them.

Hugin will do its best to align the pictures for you, and its result is either a near-perfect rendering or a misshapen mess. If the mess is from a minor mistake, you can correct it, but for serious errors such as insufficient overlap or bad focus, you will need to do a new photography session. With plenty of overlap, it sometimes is enough to simple delete the offending photographs and let the others fill in that part of the image.

Figure 8 shows the complete, but reduced resolution, die photograph that I took of the Clipper Chip. This was built from 1,475 surface photographs that were stitched together by Hugin.

9.5 Further Reading

While you should get a proper chemistry education for its own sake, textbooks on chemistry as written for chemists don't cover these sorts of procedures. Instead, you should pick up books on Failure Analysis, which can double as coffee table books for their nifty photographs of disassembled electronics.

After mastering surface photography, there are all sorts of avenues for continuing your new hobby. Using polishing equipment or hydrofluoric acid, you can remove the layers of the chip in order to photograph its internals. The neighbors at the Visual6502 project took this so far as to work backward from photographs to a working gate-level simulation in Javascript!

Additionally, you can decap a chip while it's still functional to provide for invasive or semi-invasive attacks. For invasive attacks, take a look at Chris Tarnovsky's lectures, as he's an absolute master at sticking probe needles into a die in order to extract firmware. Dr. Sergei Skorobogatov's Ph.D. thesis describes a dozen tricks for semi-invasively shining lasers into chips in order to extract their secrets, while Dmitry Nedospasov's upcoming thesis is also expected to be nifty.

Neighborly thanks are due to Andrew Righter and everyone who was polite enough not to yell at me for the die photos that I posted with improper exposure or incomplete decapsulation.

Cheers from Samland,

—Travis

[16] For fancy things like recovering gates in delayered chips, more sophisticated software is needed, but panorama software suffices when only the top layer is being photographed.

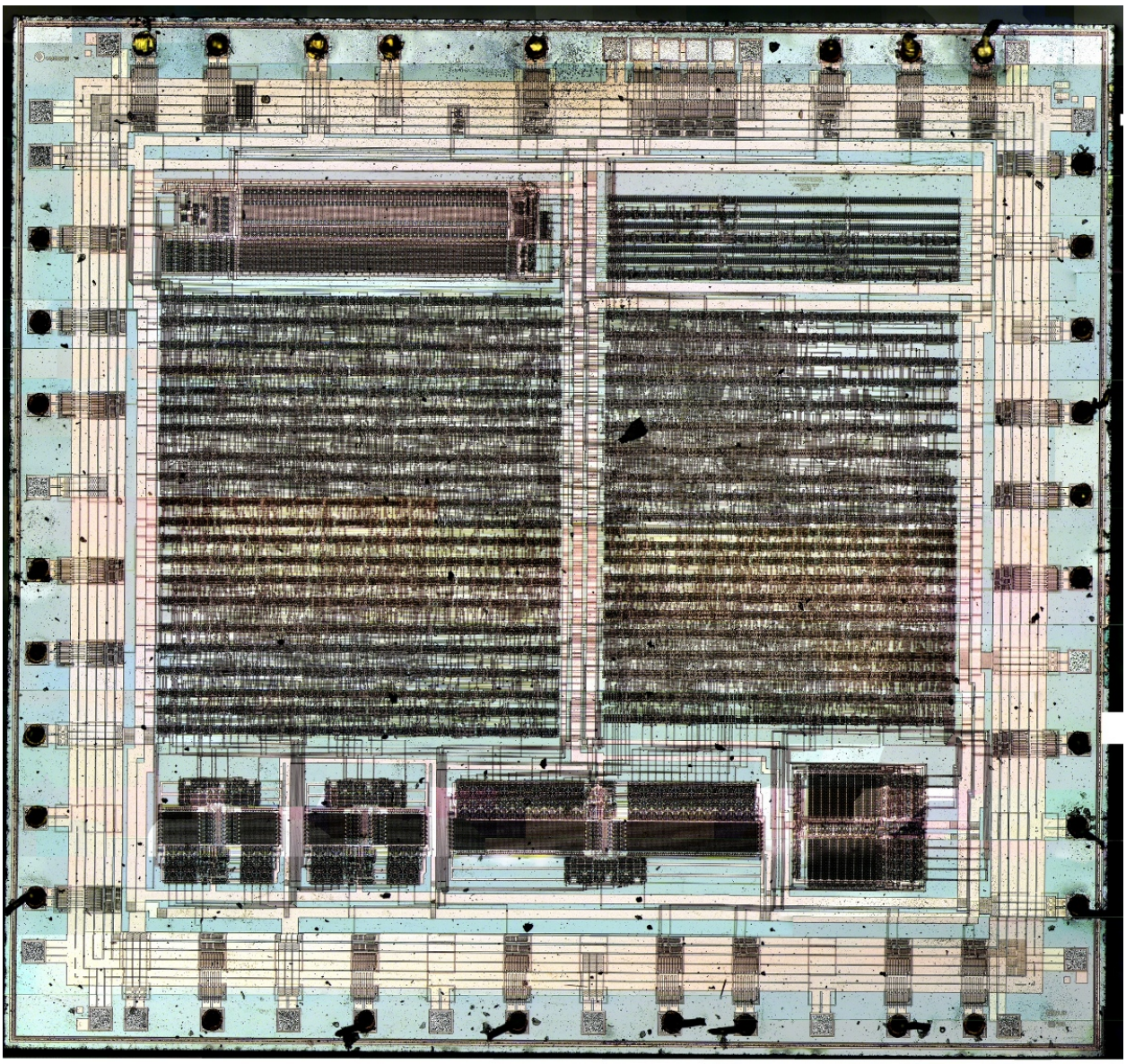


Figure 8: This is the complete die photograph of the Clipper Chip at reduced resolution.

10 Forget Not the Humble Timing Attack

by Colin O'Flynn

Judge not your neighbour's creation, as you know not under what circumstances they were created. And as we exploit the creations of those less fortunate than us, those that were forced to work under conditions of shipping deadlines or unreasonable managers, we give thanks to their humble offering of naïve security implementations.

For when these poor lost souls aim to protect a device using a password or PIN, they may choose to perform a simple comparison such as the following.

```
int password_loop(void){
  unsigned char master_password[6];
  unsigned char user_password[6];

  read_master_password_from_storage(master_password);
  wait_for_pin_entry(user_password);

  for (int i = 0; i < 6; i++){
    if (master_password[i] != user_password[i]){
      return 0;
    }
  }
  return 1;
}
```

Which everyone knows are subject to timing attacks. Such attacks can be thwarted of course by comparing a hash of the password instead of the actual password, but simple devices or small codes such as bootloaders may skip such an operation to save space.

10.1 A PIN-Protected Hard Drive

Let's look at a PIN-protected hard drive enclosure, which the vendor describes as a "portable security enclosure with 6 digit password." This enclosure formats the hard drive into two partitions, the Public partition and the secured Vault partition. The security of the Vault is entirely given by sacrilegious changes to the partition table, such that if you remove the hard disk from the enclosure and plug into a computer the OS won't recognize the disk, thinking it tainted. The data itself is still there however.

The PCB contains four ICs of particular interest: a Marvell 88SA8040 Parallel ATA to Serial ATA bridge, a JMicron JM20335 USB to PATA bridge, a WareMax WM3028A (no public information), and a SST 39VF010 flash chip connected to the WM3028A. There's also a number of discrete logic gates including two 74HCT08D AND devices and one 74HC00D NAND device. These logic gates are used to multiplex multiple parts from apparently limited IO pins of the WM3028A. It would appear that the system passes the Parallel ATA data through the WM3028A chip, which is presumably some microcontroller-based system responsible for fixing reads of the partition table once the correct password is put in.

The use of discrete logic chips for multiplexing IO lines ultimately makes our life easier. In particular one of the 74HCT08D chips, U10, provides us with a measurement point for determining when the password has failed the internal test.

Pin 3 of the switch is the multiplexing pattern from the microcontroller. Remember we must determine when the microcontroller has read the pin, not simply when the user pushed the pin. Knowing that this button was pressed, and thus caused the 'Wrong PIN' LED to come on, we can measure the time between when the microcontroller has read in the entire PIN and when the LED goes on.

We then break the system one digit at a time by measuring the time after the last button is pressed. First we enter 0-6-6-6-6-6, then 1-6-6-6-6-6, 2-6-6-6-6-6, etc. The delay between reading the button press and

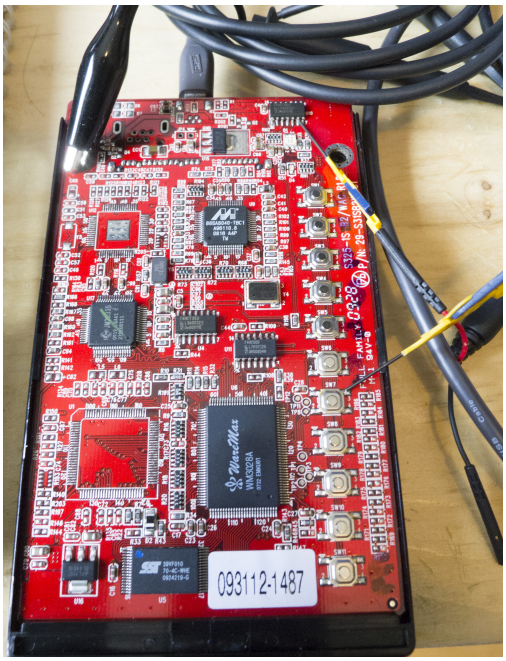


Figure 9: Pin-Protected Hard Disk

displaying the LED will be shortest if the first digit is wrong, longer if the first digit is right. A moving-picture version of this is available on the intertubes.[17]

An example of the oscilloscope capture of this is shown in Figure 10, where the correct password is 1-2-3-4-5-6. Note the jump in time delay between 0-6-6-6-6-6 and 1-6-6-6-6-6. This continues for each correct digit. Thus for a 6-digit pin, we guess only a worst case of $10 * 6 = 60$ options, instead of the million that would be required for brute-forcing the full pin.

10.2 TinySafeBoot for the Atmega328P

But what if the clever developer decided to not tell the user when they've entered a wrong password? A security-conscious bootloader might wish to avoid being vulnerable to timing attacks, but is attempting to avoid adding hash code for size reasons. An example of this is pulled from a real bootloader which has a password feature. When a wrong password is entered jumps into an endless loop, effectively avoiding providing information that would be useful for a timing attack.

In particular, let's take a look at TinySafeBoot, which is a very small bootloader for most AVR microcontrollers.[18] This wonderful bootloader has many features, such as using a single IO pin, auto-calibrating baud rate, and automatically build a bootloader image for you. And, as already mentioned, it contains a password feature.

But compare the measurements of the power signatures shown in Figure 11, which is the bootloader running on an AtMega328P. The correct password is `{0x61, 0x52, 0x77, 0x6A, 0x73}`. If we measure the power consumption of the device, we observe clear differences between the correct and incorrect guesses. This can be done by using a resistor in-line with the microcontroller power supply, such as by lifting a TFQP package pin.

The code for the password feature looks as in the following listing. Note when you receive an incorrect

[17] `http://tinyurl.com/pintiming`

[18] You can find more information about this bootloader at `http://jtxp.org/tech/tinysafeboot_en.htm`.

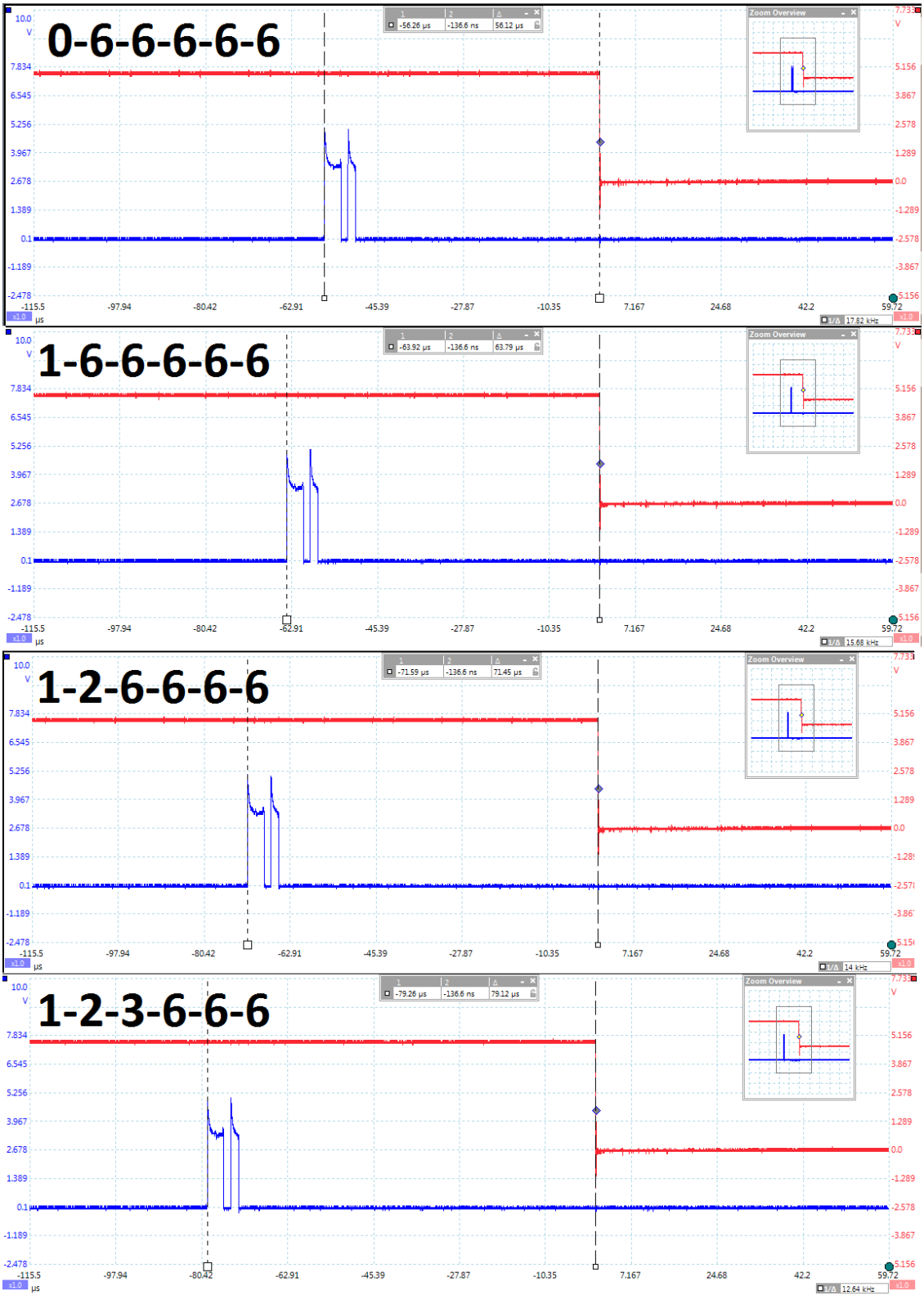


Figure 10: Timing Results

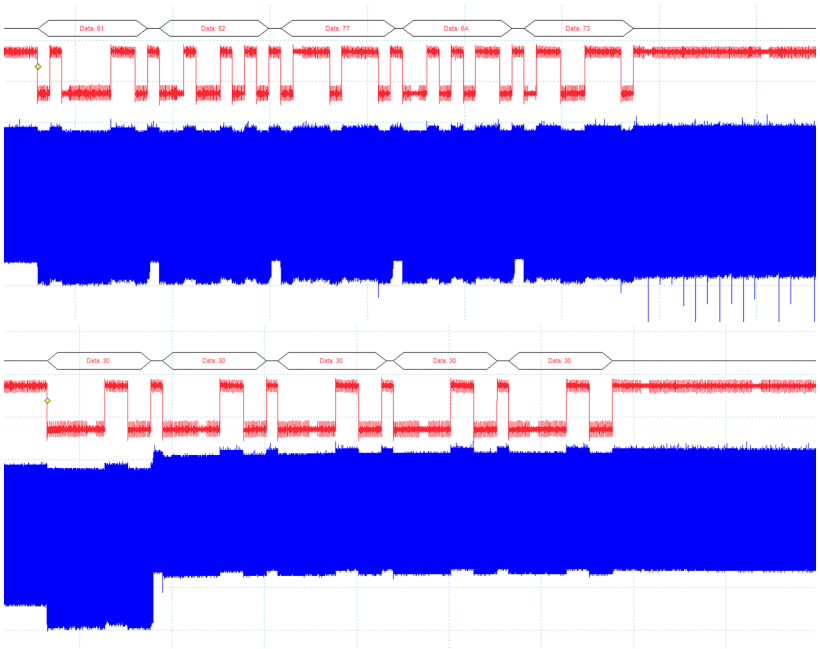


Figure 11: Power Analysis. Above is a correct guess, Below is incorrect.

character the system jumps into an infinite loop at the `chpwl` label, meaning a reset is required to try another password.

```
CheckPW:
chpw1:
        lpm tmp3, z+                    ; load character from Flash
        cpi tmp3, 255                   ; byte value (255) indicates
        breq chpwx                      ; end of password -> exit
        rcall Receivebyte               ; else receive next character
chpw2:
        cp tmp3, tmp1                   ; compare with password
        breq chpw1                      ; if equal check next character
        cpi tmp1, 0                     ; or was it 0 (emergency erase)
chpwl:  brne chpwl                      ; if not, loop infinitely
        rcall RequestConfirmation       ; if yes, request confirm
        brts chpa                       ; not confirmed, leave
        rcall RequestConfirmation       ; request 2nd confirm
        brts chpa                       ; cannot be mistake now
        rcall EmergencyErase            ; go, emergency erase!
        rjmp Mainloop
chpa:
        rjmp APPJUMP                    ; start application
chpwx:
;       rjmp SendDeviceInfo             ; go on to SendDeviceInfo
```

We can immediately see the jump to the infinite loop in the power trace! It happens as soon as the device receives an incorrect character of the password. Thus despite the original timing attack failing, with a tiny bit of effort we again find ourselves easily guessing the password.

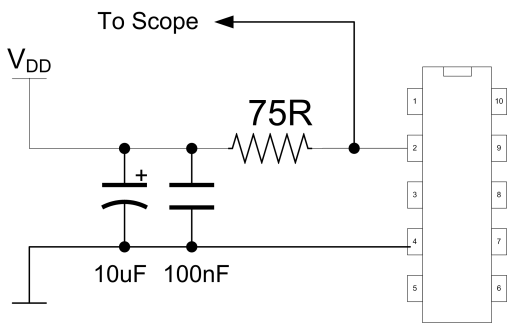


Figure 12: Tapping VCC for Power Analysis

Measuring the power consumption of the microcontroller requires you to insert a resistor into the power supply rail. Basically, this requires you to perform the schematic as shown in Figure 12. Note you can insert it either into the VCC or the GND rail. It may be that the GND rail is cleaner for example, or it may be that it's easier to physically get at the VCC pin on your device.

For a regular oscilloscope you may need to build a Low Noise Amplifier (LNA) or Differential Probe. I've got some details of that in my previous talk and whitepaper.[19] You can expect to make a probe for a pretty low cost, so it's a worthwhile investment!

In terms of physically pulling this off, the easiest option is if you build a breadboard circuit with the AVR and a resistor inserted in the power line. Be sure you have lots of decoupling after the resistor, which will give you a much cleaner signal. If you're looking to use an existing board, you can make a 'cheater' socket with a resistor inline, as in Fig B, which was designed for an Arudino board.

Real devices are likely to be SMD. If you're attacking a TQFP package, you might find it easiest to lift a lead and insert a 0603 or 0402 resistor inline with the power pin. You might wish to find a friendly neighbour with a steady hand and a stereo microscope for this if you aren't of strong faith in your soldering!

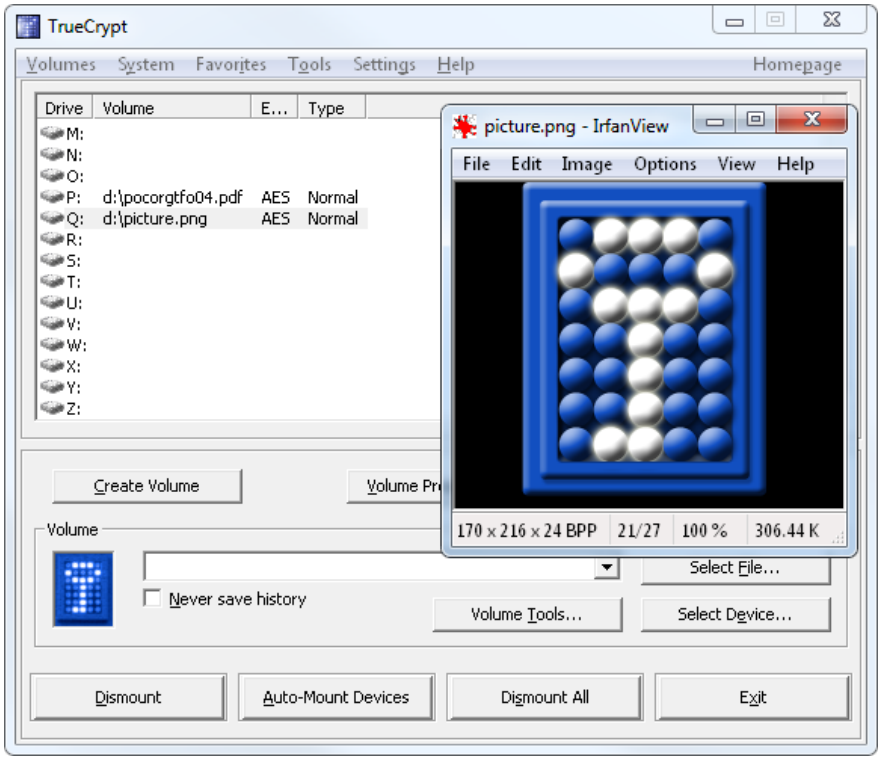
Thus when attacking embedded systems, the timing attacks often present a practical entry method. Be sure to carefully inspect the system to determine the 'correct' measurement you need to use, such as measuring the point in time when the microcontroller reads an I/O pin, not simply when an external event happens.

When designing embedded systems, store the hash of the users password, lest ye be embarrassed by breaks in your device.

[19] http://newae.com/blackhat

11 This Encrypted Volume is also a PDF; or, A Polyglot Trick for Bypassing TrueCrypt Volume Detection

by Ange Albertini



In this article I will show you a nifty way to make a PDF that is also a valid TrueCrypt encrypted volume. This *Truecryption* trick draws on *Angecryption* from PoC||GTFO 03:11, so if you missed it you can go back in PoC-time now or later, and enjoy even more common file format schizophrenia!

11.1 What is TrueCrypt?

If you open a TrueCrypt container in a hex editor, you'll see that, unlike many binary formats, it looks like entirely random bytes. It does in fact have a header that starts with the magic signature string `TRUE` at file offset `0x40`, but this header is stored encrypted, and thus you can't spot it offhand. To decrypt the header, one needs both the correct password and the hopefully random salt that is stored in the bytes 0-63, just before the encrypted header.

So, a TrueCrypt file starts with 64 bytes of randomness, used as salt to derive the *header key* from the password. This key is used to decrypt the header. If the result of the decryption starts with `TRUE`, then it means the password was correct, and the now decrypted header is parsed further. In particular, this header contains *volume keys*, which are, in turn, used to encrypt/decrypt the blocks and sectors of the encrypted drive.

Importantly, the salt itself is only used to decrypt the header. This is to defend against rainbow table-like precomputing attacks.

Let's start with an existing TrueCrypt volume file for which we know the password. We are not going to change its actual contents or the header's plaintext, but we are going to re-encrypt the header so that the whole becomes a valid PDF file while remaining a valid TrueCrypt volume as well.

Because the salt is supposed to be random, it can be anything we choose. In particular, it can double as any other file format's header. Using the original salt and password, we can decrypt the header. By choosing a new salt—which starts with the header of our new binary target—we derive new keys, and can thus re-encrypt the header to match our new salt.

So, our new file contains the new salt, the re-encrypted header, and the original data sectors of the TrueCrypt container. But where will the new PDF binary content go?

For merging in the new content, we are going to use the trick that the readers of *Angecryption*, PoC‖GTFO 03:11, must have guessed already. As we showed there, in many binary formats such as PDF, PNG, etc., it is possible to reserve a big chunk of space filled with dummy data right after the format's header, and have the binary format's interpreters simply skip over that chunk. This is exactly what we are going to do: all of the TrueCrypt volume data would go into the dummy chunk, followed by the new binary content.

If we want a valid binary file to be a TrueCrypt polyglot, we must fit its header and the declaration for the dummy chunk within 64 bytes, the size of the salt. For *Angecryption*, we managed with only 16 bytes to play with, so having 64 bytes almost feels like sinful and exuberant waste.

11.2 An elegant PDF integration

So far, our PDF/TrueCrypt polyglot looks like no contest. To add a bit of challenge, let's make it with standard PDF-making tools alone. We'll ask `pdflatex(1)` nicely to include the TrueCrypt volume into our polyglot.

Specifically, we'll create a dummy stream object directly inside the document, using the following `pdflatex` commands:

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {pocorgtfo/truecryption/volume}
\endgroup
```

The bytes between the start of the resulting PDF file and our object that contains the TrueCrypt container will depend on the PDF version and its corresponding structure. Luckily, the size of this PDF head-matter data is typically around `0x20`, well below `0x40`. Plenty of legroom on this polyglot flight!

So our PDF will start with its usual header, followed by this standard stream object we created to play the role of a dummy buffer for the TrueCrypt data. We now need to readjust the contents of this buffer so that the encrypted TrueCrypt header matches its salt, which contains the PDF header, and we then get a standard PDF that is also a TrueCrypt container.

11.3 Conclusion

This technique can naturally be applied to any other file format where we can fit the header and a dummy space allocation within its first 64 bytes, the size of TrueCrypt's initial salt.

Moreover, inserting your encrypted volume into a valid file—while keeping it usable—also has the benefit of putting it under the radar of typical TrueCrypt detection heuristics. These heuristics rely on encrypted TrueCrypt volumes having a round file size, uniformly high entropy, and no known header present. Our method breaks all of these heuristics, and, on top of that, leaves the original document perfectly valid and plausibly deniable.[20]

[20] *Of course, this advice is legally worth exactly what you paid for it, and likely less. No warranty intended or implied, void where prohibited by law, etc., etc., etc. Not endorsed by any lawyers real, imaginary, or played-on-TV, but may be considered "digital cyber-bullets" by some. You may be called a merchant of digital cyber-polyglot death, too—you have been warned!* –PML

12 How to Manually Attach a File to a PDF

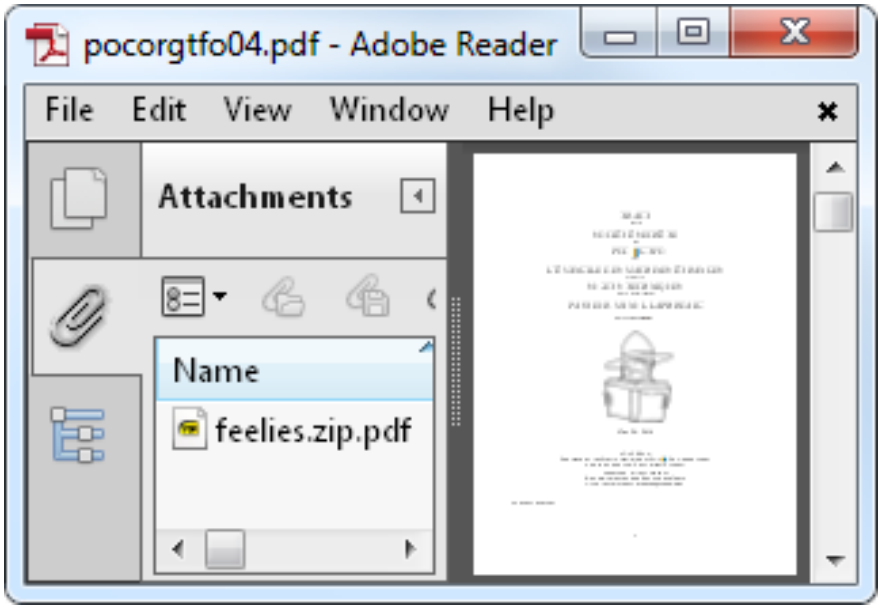
by Ange Albertini

If you followed the PoC||GTFO's March of the Polyglots to date, you may have noticed that until now the feelies were added in a dummy object at the end of the PDF document. That method kept `unzip(1)` happy, and Adobe PDF tools were none the wiser.

Yet Adobe in its wisdom created its own way of attaching files to a PDF!

> One of the great features of PDF is its ability to carry attached files, just as e-mail messages can carry attached files. Any kind of file, and any number of files, can be sucked into a PDF file. These are held internal to PDF as "stream" objects, one of the basic 8 object types from which all PDF content is built (numbers, arrays, strings, true, false, names, dictionaries and streams). Streams start with a dictionary object but then carry along an arbitrarily long sequence of arbitrary 8-bit bytes. Stream objects meet the generic description for disk files quite well.
>
> —Jim King at Adobe

So, dear reader, prepare to be sucked in into PDF feature(creep) greatness![21]



Of course, we could use Adobe software to attach the feelies, but this is not the Way of the PoC. Instead, we'll use our trusty `pdflatex(1)`.

Pdflatex allows us to directly create our own PDF objects from the TeX source, whether they are stream or standard objects. For Adobe tools to see a PDF attachment, we need to create 3 objects:

- the stream object with the attached file contents;
- a file specification object with the filename used in the document;
- an annotation object with the /FileAttachment subtype.

[21] *Some alarmist neighbors predict that the Universe will gravitationally collapse upon itself due to uncontrolled PoC||GTFO expansion. Fear not, neighbors: an international action on PoC footprint is coming! On a second thought, though, since you are all Merchants of Dire PoC now, maybe fear twice as hard? –PML*

There are a couple of things to keep in mind. First, Adobe Reader refuses to extract attachments with a ZIP extension, so we'll need to use a different one. For the plain old `unzip` still to work on the resulting PDF file (after a couple of fixes), we must make sure our attachment is stored in the PDF byte-for-byte, without additional PDF compression.

Here is the code we need. Note that after creating our PDF objects, we can refer to them via `\pdflastobj`; to output the actual value, we prepend that reference with the `\the` keyword.

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        attr {/Type /EmbeddedFile}  file {feelies.zip}
  \immediate\pdfobj{<<
        /Type /Filespec   /F (feelies.zip.pdf)  /EF <</F \the\pdflastobj\space 0 R>>
    >>}
  \pdfannot{
        /Subtype /FileAttachment  /FS \the\pdflastobj\space 0 R
        /F 2 % Flag: Hidden
    }
\endgroup
```

Finally, for some reason Adobe software fails to see an annotation object when it's the last one in the file. To work around this, we'll just have to make sure we have some text after that object.

12.1 Increasing compatibility

Sadly, after we use this method and put the (extension-renamed) ZIP into PDF as a standard attachment, plain old `unzip` will fail to unpack it. To `unzip`, the file doesn't look like a valid archive: the actual ZIP contents are neither located near the start of the file (because it's a TrueCrypt polyglot) nor at the end (because our document is big enough so the XREF table is bigger than the usual 64Kb threshold). Let's help `unzip` to find the ZIP structures again!

Luckily, this is easy to do. All we need is to duplicate the last structure of the ZIP file—the End of Central Directory—which points to the body, the Central Directory. This structure is just 22 bytes long, so it won't make a big difference. When duplicating, we change the offset to the Central Directory so that it's pointing to the correct place in the PDF body. We then need to adjust the offsets in each directory entry so that our files' data is still reachable—and voilà, we have an attachment that is visible both to the fancy Adobe tools and to the good old classic `unzip`!

13 Ode to ECB

by Ben Nagy

Oh little one, you're growing up
You'll soon be writing C
You'll treat your ints as pointers
You'll nest the ternary
You'll cut and paste from github
And try cryptography
But even in your darkest hour
Do not use ECB

CBC's BEASTly when padding's abused
And CTR's fine til a nonce is reused
Some say it's a CRIME to compress then encrypt
Or store keys in the browser (or use javascript)
Diffie Hellman will collapse if hackers choose your g
And RSA is full of traps when e is set to 3
Whiten! Blind! In constant time! Don't write an RNG!
But failing all, and listen well: Do not use ECB

They'll say "It's like a one-time-pad!
The data's short, it's not so bad
the keys are long—they're iron clad
I have a PhD!"
And then you're front page Hacker News
Your passwords cracked—Adobe Blues.
Don't leave your penguin showing through,
Do not use ECB

Sometimes it can seem like there's ECB everywhere. ECB on TV, ECB in music, it's endless. But that doesn't make it safe. Or right. So tune out and avoid ECB, no matter what your friends, the TV, or your favourite cryptographer tells you.

Canadian Joke Council

True Bugs Wait ♡

@natashenka
#truebugswait

14 A Call for PoC

by Pastor Manul Laphroaig

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

14.1 PoC Contributions

Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, or German.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me something about file formats that even Ange Albertini doesn't already know; teach me how to make an image that's invisible at high resolution but at low resolution is exposed by dithering; or, teach me that an old exploitation trick still works on QNX. Show me how to emulate Atlas's RFCat as a GNURadio block. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

You can expect PoC||GTFO 0x05, our sixth release, to appear in print soon at a conference of good neighbors. We've not yet decided whether to include crayons, but you can be damned sure that it'll be a good read.